Behavioural Categoricity of Abstract Data Type Specifications

Pierre Lescanne
Centre de Recherche en Informatique de Nancy, Campus Scientifique: BP 239, F54 506 Vandoeuvre-les-Nancy Cedex, France

In this note we want to present the concept of behavioural categoricity of an abstract data type specification. Intuitively, a specification is behaviourally categoric if it captures the external views the user can have on the data type. More specifically, using this specification, it is possible to prove that two objects are equal if and only if they behave the same, or informally speaking, if and only if they implement the same black box. Providing a general algorithm for proving the behavioural categoricity of any specification is impossible because that algorithm could also decide whether a finite presentation of a group presents the trivial group or not, which Rabin proved to be undecidable. We show by an example of a specification of circular lists that the proof of the categoricity must be done carefully.

In this note we want to present the concept of behavioural categoricity of an abstract type specification. Intuitively, a specification is behaviourally categoric if it captures all the external views the user can have on the data type. More specifically, using this specification, it is possible to prove that two objects are equal if and only if they behave the same, or informally speaking, if and only if they implement the same black box. Providing a general algorithm for proving behavioural categoricity of any specification is impossible, because that algorithm could also decide whether a finite presentation of a group presents the trivial group or not, which Rabin proved to be undecidable. We show by an example of a specification of circular lists, that the proof of the categoricity must be done carefully.

SYNTACTIC EQUALITY AND BEHAVIOURAL EQUALITY

We assume the reader has knowledge of the algebraic specification of abstract data types.[1,2] Let us present and clarify some terminology. First, the environment is a set of type operations and equations that are called primitive or external. For simplicity, we suppose they are infinitely many equations, i.e. a congruence written $=^e$ here, defining a primitive heterogeneous algebra. In general, an algebraic specification is divided into two parts. The first part describes the functionality or signature of the operations, these operations are *observers* that yield a result of an external type, or *internal operations* that yield a result of the type of interest. Objects are described by composition of internal operations. The representation is not unique, but up to a congruence generated by the axioms. A *compound observer* is a term whose outermost operation is a simple observer (sometimes a compound observer is called an observer if there is no ambiguity; otherwise we use the name simple observer, to make a clear distinction). A *basis* is a minimal generator subset among the internal operations, that means it is sufficient to generate all the objects of the abstract data type; its elements are called *constructors*. All other elements are called *extensions*. Terms that contain only constructors are often called *normal forms*.

The second part of the specification contains the *axioms*. They are equations that specify the abstract data type by describing the relations between the constructors, and by defining the extensions and the observers. Two ground terms are said to be equivalent (for the axioms) if it is possible to transform one to the other by successive replacements of equal by equal.

Since variables can be of external or internal type, let us call an *internal-variable-free-extension* a term built from only internal operations and external variables. If in addition it has an observer as outermost operation it is called an *internal-variable-free-compound-observer*. The main problem about specification is to know whether it actually defines the extensions and the observers in terms of constructors. A specification is *well-defined*:

1. If every internal-variable-free-extension can be proved equivalent to a unique term (uniqueness is modulo the relation between constructors) which contains only constructors and external variables. This property is sometimes called the *normal form lemma*.
2. If every internal-variable-free-compound observer can be proved equivalent to a unique external variable term. The existence of such an external variable term is sometime called *sufficient completeness*. The uniqueness is sometime called *relative consistency* because it means that every primitive terms are equal in the specification if and only if they are also in the primitive one.

For the sake of simplicity, we will assume in this paper that all the operations are total, but we guess that the concepts defined here should be easily generalized to the case of partial operations.

Now we describe the two main approaches to the semantics of the abstract data types. Both refer to an *abstract model* which is an initial or final algebra according to the semantics we use. The classes of the congruences which are compatible with a well defined specification, and which are extensions of the primitive congruence, constitute a 'lattice' for the inclusion with a minimum (or initial) element and a maximum (or final) element.

The *initial algebra* semantics[3] is proof theory-oriented; it considers that two objects are equivalent if and only if

CCC-0010-4620/83/0026-0289 $02.00

it can be *proved* that they are. In this case, we suggest calling the 'initial algebra semantic equivalence' syntactical equivalence (written $=^s$), that is, equivalence deduced by reasoning on terms from the axioms. This approach can be seen as syntactic, since, first $=^s$ is syntactically defined, and secondly the abstract model, called the *initial algebra*, is the quotient of the term algebra under the equivalence relation $=^s$. Thus, this semantics is strongly related to the algebraic specification. Notice that relative consistency asserts that axioms do not introduce relations among 'external objects', which means that $=^e$ and $=^s$ coincide on external terms.

The final algebra semantics is user-oriented.[4,5] It considers that two objects are different if there exist some observers (simple or compound) that yield different values when applied to each of the two objects. In other words, two objects are equal in the abstract model if, for an outside user, they behave the same. This approach is more semantic, since it considers objects only from the behavioural point of view. The behavioural equivalence[6] (written $=^b$), is the equivalence between two objects which have the same behaviour under any observer. Notice that syntactic equivalence implies behavioural equivalence. The abstract model is a *final algebra*. It is the quotient under behavioural equivalence of any algebras, one of which can be the initial algebra. All these final algebras are isomorphic and we say 'the' final algebra. This semantics depends on the specification of the observers.[7]

BEHAVIOURAL CATEGORICITY

A specification is behaviourally categoric if the syntactic equivalence coincides with the behavioural equivalence; thus the specification is powerful enough to capture syntactically the whole behaviour of the objects. The archetype of a non-behaviourally categoric specification of a type is *Set* (Fig. 1). The two sets INSERT(INSERT(EMPTY( ), *a*), *b*) and INSERT(INSERT(EMPTY( ), *b*), *a*) are behaviourally equivalent but not syntactically equivalent. Notice that *Set* could be made behaviourally categoric by adding the axioms

INSERT(INSERT(EMPTY ( ), *a*), *b*) = = INSERT (INSERT(EMPTY ( ), *b*), *a*) and
INSERT(INSERT (*s*, *a*), *a*) = = INSERT(INSERT (*s*, *a*).

Type *Set* [*Item*, *Bool*] **where** *Item* **has** EQ: (*Item*, *Item*) → *Bool*

Operations
EMPTY: ( ) → *Set*
INSERT: (*Set*, *Item*) → *Set*
HAS: (*Set*, *Item*) → *Bool*

Axioms
HAS(INSERT(*e*, *a*), *b*) = = IF EQ(*a*, *b*) THEN TRUE ELSE HAS(*e*, *b*)
HAS(EMPTY ( ), *a*) = = FALSE.

Figure 1. Specification of set.

Thus, we try to prove behavioural categoricity to show that we have not accidentally left out some of the equations necessary for specifying the type by the initial algebra method. However, we may state the following theorem.

Theorem 1

There exists no general algorithm, on input of any abstract data type specification, that decides whether this specification is behaviourally categoric or not.

Proof. If such an algorithm exists it can decide the behavioural categoricity of a very special family of abstract data types: the finitely presented groups. These are abstract data types with the three classical operations of group theory, namely $*$, $^{-1}$, and E, plus a finite set of constants which we call generators. The axioms are the axioms of groups plus a finite number of equations between terms, which we call the relations (Fig. 2). The group that is said to be presented by these generators and these relations is the initial algebra. Because there are neither extensions nor observers, notice that first, the specification is trivially well-defined, and secondly, all the terms are behaviourally equivalent. Thus, the final algebra is the trivial group with carrier (E). Rabin [Ref. 8, Theorem 2.2] showed that the problem of proving that a finitely presented group is trivial is undecidable. □

Type *Group*

Operations
Group * *Group* → *Group*
(*Group*)$^{-1}$ → *Group*
E, $A_1, \ldots, A_n$: ( ) → *Group*

Axioms
$(x * y) * z = = x * (y * z)$
$E * x = = x$
$x^{-1} * x = = E$
$w_1 = = w_1'$
...
$w_m = = w_m'$

Figure 2. Group with n generators $A_1, \ldots, A_n$ and m relations.

THE CIRCULAR LIST EXAMPLE

In this section we describe a data type which occurs frequently in computer systems, the circular lists (Fig. 3, see Ref. 9 for their use).

Theorem 2

If *Item* contains at least two elements, i.e. if the external congruence $=^e$ is not identically true, then the abstract data type *Circular_List* [*Item*] is behaviourally categoric.

Before proving this theorem, notice two facts. First, although the property on the number of elements of *Item*

Type *Circular_List* [*Item*]
Operations
INIT: (*Item*) → *Circular_List*
INSERT: (*Item*, *Circular_List*) → *Circular_List*
ROT: (*Circular_List*) → *Circular_List*
WINDOW: (*Circular_List*) → *Item*

Axioms
ROT(INIT(a)) = = INIT(a)
ROT(INSERT(a, INIT(b))) = = INSERT(b, INIT(a))
ROT(INSERT(a, INSERT(b, cl))) = = INSERT(b, ROT(INSERT(a, cl)))
WINDOW(INIT(a)) = = a
WINDOW(INSERT(a, cl)) = = a.

Figure 3. Specification of Circular_List.

is a little odd, it will become clearer why we need it when we use these elements to discriminate among values of *Circular_List*. Secondly, the proof of the categoricity is not straightforward, as we will show.

Let us now remark that the axioms defining *Circular_List* can be transformed to a confluent and Noetherian rewriting system where = = is changed to = >. Therefore, this rewriting system can be used to decide the syntactic equivalence of terms:[10] two terms are syntactically equivalent if their normal forms are equal (more formally, we have $=^s$ equivalent to = >*. < =*). To prove Theorem 2, we have to prove that two different normal forms can be discriminated by compound observers. Let us adopt the following notations.

$$\text{INSERT}(a_{n-1}, \ldots, \text{INSERT}(a_1, \text{INIT}(a_0)) \ldots) = \langle a_{n-1}, \ldots a_0 \rangle = \mathbf{a}$$
$$\text{INSERT}(b_{m-1}, \ldots, \text{INSERT}(b_1, \text{INIT}(b_0)) \ldots) = \langle b_{m-1}, \ldots, b_0 \rangle = \mathbf{b}$$

Let us sketch the proof. Because there are no axioms, the initial algebra for INIT and INSERT is just $Item^+$ (non-empty sequences of items) with $\text{INIT}(a) =^s \langle a \rangle$ and $\text{INSERT}(a, \langle b_{m-1}, \ldots, b_n \rangle) =^s \langle a, b_{m-1}, \ldots, b_0 \rangle$ (essentially Lemma 1). Then the initial algebra for the type with ROT and WINDOW has the same carrier because $\text{ROT}(\langle a_{n-1}, \ldots, a_0 \rangle) =^s \langle a_{n-2}, \ldots, a_0, a_{n-1} \rangle$ and $\text{WINDOW}(\langle a_1, \ldots, a_n \rangle) =^s a_1$ (Lemmas 2, 4, 5). That means INIT and INSERT form a basis. Then if

$$\mathbf{a} =^s \langle a_{n-1}, \ldots, a_0 \rangle \sim =^s \langle b_{m-1}, \ldots, b_0 \rangle =^s \mathbf{b}$$

either there exists a j with $a_j \sim =^e b_j$ in which case

$$\text{WINDOW}(\text{ROT}^{j-1}(\mathbf{a})) \sim =^s \text{WINDOW}(\text{ROT}^{j-1}(\mathbf{b})),$$

or **a** is a prefix of **b** (or vice versa), say **b** = **a**. **d**. **c**. In that case, choose $f \sim =^e d$ and,

$$\mathbf{d} =^s \text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT}(b, \mathbf{b}))) \sim =^s \text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT}(b, \mathbf{a}))) =^s b$$

Lemma 1. Well-Definedness Lemma

1. The normal forms are $\text{INSERT}(a_{n-1}, \ldots, \text{INIT}(a_0))$. More precisely, they are either $\text{INIT}(a)$ or $\text{INSERT}(a, t)$ where t is a normal form.
2. $\text{WINDOW}(t)$ is equal to a variable, where t is an internal-variable-free term.

Proof. By induction on the complexity of terms: here complexity is given by the following function c:

$$c(\text{ROT}(t)) = (c(t))^2$$
$$c(\text{INSERT}(a, t)) = c(t) + 1$$
$$c(\text{INIT}(a)) = 1. \quad \square$$

Lemma 2

$\text{ROT}(\langle a_{n-1}, \ldots, a_0 \rangle) = >^* \langle b_{n-1}, \ldots, b_0 \rangle$ where $b_1 =^e a_{i-1}$ if $0 < i < n$ and $b_0 =^e a_{n-1}$.

Proof. By induction on n.

Lemma 3

If $0 < k < n$. $\text{ROT}^k(\langle a_{n-1}, \ldots, a_0 \rangle)^* \langle b_{n-1}, \ldots, b_0 \rangle$ where $b_i = a_{i-k}$, if $n > i > = k$ and $b_i = a_{n-k+i}$, if $k > i > = 0$.

Proof. We prove the result by induction on k. If $k = 0$, it is straightforward. Suppose the result is true for k, then compute the value for $k = 1$. By induction:

$$\text{ROT}^{k+1}(\langle a_{n-1}, \ldots, a_0 \rangle) = >^* \text{ROT}(\langle c_{n-1}, \ldots, c_0 \rangle)$$

where $c_j =^e a_{j-k}$, if $n > j > = k$ and $c_j =^e a_{n-k}$, if $k > j > = 0$. By Lemma 2

$$= >^* \langle b_{n-1}, \ldots, b_0 \rangle$$

where $b_i =^e c_{i-1}$, if $n > i > = 0$ and $b_0 =^e a_{n-1}$. That is

$$b_i =^e a_{i-k+1}, \text{ if } n > i - 1 > = k \text{ and } n > i > 0$$
$$b_i =^e a_{n-k+i-1}, \text{ if } k > i - 1 > = 0 \text{ and } n > i > 0$$
$$b_0 =^e a_{n-k-1}.$$

Joining the two last assertions and simplifying the conditions, we get

$$b_i =^e a_{i-(k+1)}, \text{ if } n - 1 > i > k + 1$$
$$b_i =^e a_{n-(k+1)+i}, \text{ if } k + 1 > i > = 0.$$

or

$$b_i =^e a_{i \bmod n} \quad \square$$

Lemma 4

$\text{ROT}^n(\langle a_{n-1}, \ldots, a_0 \rangle) = >^* \langle a_{n-1}, \ldots, a_0 \rangle$.

Proof. Straightforward from Lemma 3. $\square$

Lemma 5

If $0 < k < n$, $\text{WINDOW}(\text{ROT}(\langle a_{n-1}, \ldots, a_0 \rangle) = >^* a_{n-1-k}$.

Proof. From Lemma 3.

$$\text{WINDOW}(\text{ROT}^k(\langle a_{n-1}, \ldots, a_0 \rangle)) = >^* \text{WINDOW}(\langle b_{n-1}, \ldots, b_0 \rangle) = >^* b_{n-1} \quad \square$$

where $b_{n-1} =^e a_{n-1-k}$.

The following lemma proves Theorem 2 on normal forms having the same length.

Lemma 6

$\langle a_{n-1}, \ldots, a_0\rangle =^s \langle b_{n-1}, \ldots, b_0\rangle$ if and only if $\langle a_{n-1}, \ldots, a_0\rangle =^b \langle b_{n-1}, \ldots, b_0\rangle$.

Proof. The 'if' part is trivial. Let us prove the 'only if' part:

$$\langle a_{n-1}, \ldots, a_0\rangle =^b \langle b_{n-1}, \ldots, b_n\rangle$$

implies

$$\forall k \in [1 \ldots n])\ \text{WINDOW}(\text{ROT}^k(\langle a_{n-1}, \ldots, a_0\rangle)) =^e \text{WINDOW}(\text{ROT}^k(\langle b_{n-1}, \ldots, b_0\rangle))$$

By Lemma 5, this is equivalent to

$$(\forall k \in [1 \ldots n])\ a_{n-k-1} =^e b_{n-1-k}$$

Because all a_i are equal to b_i, both circular lists are syntactically equal, this is equivalent to $\langle a_{n-1}, \ldots, a_0\rangle =^e \langle b_{n-1}, \ldots, b_0\rangle$. □

The compound observers WINDOW(ROT^k(_)) are not sufficient to discriminate between $\langle a, b\rangle$ and $\langle a, b, a, b\rangle$ or to discriminate the powers $\langle \mathbf{w}^k\rangle$ of a given circular list **w**. Likewise, they are not sufficient to discriminate among the circular lists of one element set of items.

Proof of Theorem 2. Given Lemma 6, it remains to prove the theorem for the case $n \tilde{} = m$. Suppose $m > n$ and $\mathbf{a} = \langle a_{n-1}, \ldots, a_0\rangle$ and $\mathbf{b} = \langle b_{m-1}, \ldots, b_0\rangle$, are two different normal forms, then we will show they are not behaviourally equivalent. By Lemma 5

$$\text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT}(c, \mathbf{b}))) =^s \text{WINDOW}(\text{ROT}^{n+1}(\langle c, b^{m-1}, \ldots, b_0\rangle))$$

and by Lemma 4

$$\text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT}(c, \mathbf{a}))) =^s c.$$

If $c \tilde{} =^e b_{m-n-1}$ (that is always possible because *Item* has two elements) then

$$\text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT})c, \mathbf{a}))) \tilde{} =^s \text{WINDOW}(\text{ROT}^{n+1}(\text{INSERT}(c, \mathbf{b}))).$$

Therefore **a** and **b** are not behaviourally equivalent. □

In fact, this abstract data type, with exactly these operations is also difficult to capture using final data type specifications as explained by Kamin.[7]

Acknowledgements

I would like to thank Christine Choppy, Marie-Claude Gaudel, John Guttag, Sam Kamin, Srivas Mandayam, Jean-Luc Remy and Jeannette Wing who helped me to clarify my ideas on abstract data types.

REFERENCES

1. H. A. Klaeren, Bibliography on abstract software specification. *Bulletin of the European Association for Theoretical Computer Science* **12**, 76–87 (1980).
2. R. T. Yeh, *Current Trends in Programming Methodology, Vol. 4*. Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A., 60–79 (1978).
3. 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 Ref. 2.
4. M. Wand, Final algebra semantics and data types extensions. *JCSS* **19**, 27–44 (1979).
5. V. Giarratana, F. Gimona and U. Montanari, Observability concepts in abstract data type specifications in 5th Mathematical Foundations of Computer Science, 1976, edited by A. Mazurkiewicz, *Lecture Notes in Computer Sciences* **45**, pp. 576–587, Springer Verlag (1976).
6. D. Kapur, Towards a theory for abstract data types, Ph.D. Thesis. Massachusetts Institute of Technology, MIT/LCS/TR-237, May (1980).
7. S. Kamin, Final data types and their specifications. *Trans. Programming Languages and Systems* **5**, 97–121 (1983).
8. M. O. Rabin, Recursive unsolvability in group theoretic problems. *Ann. of Math.* **67**, 172–194 (1958).
9. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* Addison Wesley, Reading, Mass. (1968).
10. G. Huet, Confluent reduction: abstract properties and applications to term rewriting systems. *J. of A.C.M.* **27**, 797–821 (1980).
11. J. V. Guttag and J. J. Horning, The algebraic specification of abstract data types. *Acta Informatica* **10**, 27–52 (1978).

Received December 1981