Fundamenta Informaticae 28 (1996) 141–152
IOS Press

The Algebraic Specifications do not have the Tennenbaum Property

Grazyna Mirkowska and Andrzej Salwicki
LITA, Université de Pau, France
e-mail: author@univ-pau.fr

Abstract. It is commonly believed that a programmable model satisfying the axioms of a given algebraic specification guarantees good properties and is a correct implementation of the specification. This convinction might be related to the Tennenbaum's property[Ten] of the arithmetic: every computable model of the Peano arithmetic of natural numbers is isomorphic to the standard model.

Here, on the example of stacks, we show a model satisfying all axioms of the algebraic specification of stacks which can not be accepted as a good model in spite of the fact that it is defined by a program. For it enables to "pop" a stack infinitely many times.

1. Introduction and Motivation

In this paper we discuss the problem of the *quality* of specifications of abstract data types. It is commonly accepted that specifications are needed during the process of creating of software. There are a few papers in the literature devoted to the question of quality. The majority of people dealing with specifications, may have an impression that any specification is good in a sense, for it specifies something. We argue that a specification may be good, better or bad. It is not enough to give a specification. It should specify *exactly* and completely what was meant to specify!

Why specifications are important? We shall use an example in which the goal of software creating is *factorized* onto two subgoals. The subgoals can be reached independently resulting in two modules of software. A specification is the only link between the modules. A *good* specification assures that the modules assembled together give a correct solution. Our example concerns the notion of inversion.[1]

Example 1.1. Let us suppose that we are to program the algorithm realising the inversion operation. What is needed? the algorithm of the inversion which uses the notions (or types, as you will) of point, line, circle and the operations as: intersection of two lines, which returns a point, intersection of two circles, etc and predicates of equality, of being parallel etc. We need a library, or better a class, which implements the notions like point, line, circle, and the basic operations such as the point of intersection of two lines, "drawing" a line through two points, etc. Having that, one can write the algorithm in a compact and clean way.

Hence we need two pieces of software: an implementing module M and another module, say, a procedure, which realizes the inversion operation in geometrical terms defined by M.

[1] *Inversion* with respect to an arbitrarily fixed circle C that has the center Q and the radius r is an operation which for a given point P returns a point P' such that: 1) the points Q, P, P' are colinear and 2) $\overline{QP} \cdot \overline{QP'} = r^2$.

We immediately realize that the module M can be used several times with different algorithms and that one algorithm can be associated with different implementing modules. In our example, the inversion algorithm may be associated with modules of planar or stereo geometry. A specification of the notions of point, circle, line and the corresponding operations is needed. It will be used by the person writing the algorithm of inversion. It will be used also by the person writing the class modules: point, circle, line and the methods as intersectn of two circles, of two lines etc. It is possible that two modules realizing the planar and the stereo geometry will satisfy the specification. □

There is a belief that a software which obeys all the properties mentioned in a specification is to be accepted since it is computable and since all the axioms listed in the specification are satisfied.

We are going to demonstrate that, *pathological, computable* models of algebraic specifications exist. These models should not be accepted in spite of the fact that they obey all the properties mentioned in the specification .

Now, let us recall the motivations and expectations for a specification of an abstract data type:

- a specification is to enable a mathematical *identification* of the class of acceptable abstract data types,
- a specification should be *complete* i.e. it should bring "the truth, the whole truth and nothing but the truth" on specified data structure,
- a specification is going to be used as an acceptance *criterion* for a piece of software that realizes it (The software may have a form of a library, a **class**[2], a package, etc. We shall refer to it as to an *implementing module*).
- a specification is going to be used as the base for analysis of algorithms operating in the environment of an implementing module (*provability* of properties of programs). It means that in a proof of a program's property we should use only the specification, we should avoid the references to the details of software's implementation of the data structure.

Later, we compare two methods of specifying data structures: the algebraic one and the algorithmic one with respect to this list.

In general, an algebraic specification [AlgSpec]is a collection of equations or Horn clauses (or a first-order formulas) in a first-order language. This type of specification is commonly used and presented in several books and articles.

The popularity of this form of specifying the data structures' properties is quite natural since the properties written as the equations are easy to understand. There is however a trap in the method. If one would like to verify whether a software is correct with respect to the algebraic specification then it is necessary to prove:

- that the data structure defined by the software satisfies the axioms of the algebraic specification and,
- that this structure determines an initial algebra (or terminal algebra) in the class of all computable structures of the same signature.

The first condition is natural and not very difficult to verify while the second one is not easy. The class of all algebras satisfying the given set of formulas may be quite rich and there are not any hints how an initial algebra looks like and how the programmer can verify whether the software created is isomorphic to the initial algebra in the corresponding class.

We submit a non-standard model M of the algebraic specification AxS of stacks. The model M is realised in the object oriented programming language Loglan'82, but *it can be rewritten in any programming language*! We present a proof that all axioms of the algebraic

[2]Yes, let us do it with a **class**!

specification are satisfied by M. We observe that M contains some pathological stacks which admit an infinite "popping". Hence the model can be rejected. However, one can construct other pathological models. Now, the problem is: how a programmer may distinguish between "good" models and "bad" ones? Can we offer a clear and simple criterion which enables to discard this one, and other unwanted models, especially, when they come in the form of software. In other words, the belief that non-standard models exist only in metamathematics, and not in real programming, needs to be revised. The programers will expect that a method will be given which enables to differentiate between the correct implementations and the incorrect ones.

We wish to point out the other problem with the algebraic specifications. A specification consists of two parts 1° the clearly visible axioms and 2° the requirement to choose the initial (or terminal) algebra in the class of all ground term algebras satisfying the given axioms. The second condition makes that there are important additional facts which are valid in the initial model and are not provable from the axioms. The problem might be stated as follow: how to extract this additional information and to join it to the visible part of the specification? How to translate it into the programmer's language?

Let $\mathfrak{L}$ be a formal language. Let Z be a set of formulas. By $Mod(Z)$ we denote the class of all models of the set Z. We call *specification of a data structure* $\mathfrak{A}$ (or of a class C of similar structures) any set of formulas Z in the language $\mathfrak{L}$ which satisfy the following properties:

- the signature of the set Z is the same as of $\mathfrak{A}$ (the one of the class C)
- $\mathfrak{A} \in Mod(Z)$ ($C \subset Mod(Z)$).

Obviously the class $Mod(Z)$ contains infinite number of different models, since,

- if there is a model M of Z then any data structure isomorphic to M is also a model of Z,

Moreover the class of all models of Z can contain several unexpected models. Hence this type of characterization is not satisfactory in several cases.

We say that a set Z of formulas is a *complete specification* of class C of similar data structures iff

- the signature of the set Z is the same as in C and
- $C = Mod(Z)$.

It means that each model of the set of axioms Z belongs to the class C and each structure from the class C is a model of the set of axioms Z.

In the language of the first-order logic one can specify any finite structure. Also some structures of importance are specified by a first-order specification e.g. the notion of Boolean algebra, of group etc. The representation theorems confirm the adequacy of the latter specifications. For example, let C_1 be the class which contain any algebra which is an isomorphic image of the algebra of subsets of a set St. The representation theorem for Boolean algebras says that the class C_1 is completely specified by the axioms of Boolean algebra. Unfortunately the language of the first-order logic is not sufficient to completely specify many data structures of importance for computer science and for mathematics. Notions of stack, of queue, of integer number do not have the adequate specifications in the language of first-order logic. Each stack, each queue and each integer is, a finite object. However, the structure of stacks (queues, integers) is infinite one. For the case of integer numbers the specification composed of two parts: the known axioms of Peano arithmetics and the requirement that the model should be computable does the job, this is the Tennenbaum's theorem[Ten].

Theorem 1.1. *If M is a recursive model of Peano's arithmetic then M is isomorphic to the standard model of natural numbers.*

For the other structures the property of Tennenbaum does not hold. On our side we propose to consider algorithmic specifications of these and other structures. [AL, cf.].

As concerns the algebraic specifications we have the following

Definition 1.1. A set of equations (or Horn clauses) Z is an algebraic specification of a data structure $\mathfrak{A}$ iff $\mathfrak{A}$ is isomorphic to the initial algebra in the class $Mod(Z)$ of all models of Z.

2. An Algebraic Specification of Stacks

As an example of an algebraic specification let us consider the structure simple but of great importance - the structure of stacks.

Definition 2.1. The abstract data structure of stacks is a relational system having the signature given below and obeying the laws AxS

$< E \cup S, push, pop, top, empty, = >$

where S is the set of stacks, E is the set of elements of stacks and *push* is a total operation and *pop*, *top* are partial operations of the following signature

$push : E \times S \longrightarrow S$

$pop : S \longrightarrow S$

$top : S \longrightarrow E$

$empty : S \longrightarrow B_o$

$= : E \times E \cup S \times S \longrightarrow B_o$

Moreover the following properties (axioms) AxS are satisfied:

Ax1 $\neg empty(push(e, s)$

Ax2 $top(push(e, s) = e$

Ax3 $pop(push(e, s)) = s$

Ax4 $\neg empty(s) \Rightarrow push(top(s), pop(s)) = s$

The relation $=$ is characterized by the usual properties of reflexivity, symmetry, transitivity and extensionality.

An initial model of these axioms will be a structure composed of the set E of elements and of all finite sequences of elements of E together with the obvious meaning of the operations: push, pop, top, empty on the sequences[ATS, Sal80]. We are going to study the question: *is any programmable model of axioms AxS the initial model of AxS?* Or equivalently, is it isomorphic to a standard model of stacks?

3. An Example of the Implementation

Our implementation is given as a class written in the Loglan'82[Loglan] programming language. It is a generic module which admits different instances of the elem type of elements of stacks. The inheritance mechanism enables to develop different extensions of the notion of element of stacks. The extensions should bring the definition of the eq a virtual function that compares two elements in such a way that the axioms of equality are satisfied.

```
unit Stacks: class;
 hidden link, e0;
 signal Stack_is_Empty, Violation;
 var e0 : elem;
 unit elem : class;
  unit virtual eq : function( e: elem): boolean;
   (* it is assumed here that any application of the class Stacks will redefine the type of
elem e.g. by inheritance, and will provide a method of comparing two elem objects in such a
way that the axioms of equality will be satisfied *)
   end eq;
end elem;
unit link : class (el:elem);
  var prev : link;
end link;
unit stack : class;
  var top : link;
begin
```

```
  if not this stack is extra and not this stack is normal
  then
    raise Violation
  fi;
end stack;
unit extra: stack class;
end extra;
unit normal : stack class;
end normal;
unit empty : function(s:stack) : boolean;
begin
  result := (s is normal) and (s.top=none)
end empty;
unit top function(s:stack): elem;
begin
  if empty(s) then
   raise Stack_is_empty
  else
   if (s is normal) then
     result := s.top.el
  else
    if not s.top=none then result := s.top.el else result := e0 fi;
  fi;
 fi;
end top;
unit pop : function(s : stack) : stack;
begin
  if empty(s) then
    raise Stack_is_Empty
  else
     if (s is normal) then
        result := new normal;
        result.top :=s.top.prev
     else
       result := new extra;
       if not s.top=none
       then
         result.top := s.top.prev
      fi
    fi
  fi;
end pop;
unit push : function( e: elem, s: stack): stack;
begin
  if ( s is normal) then
    result := new normal else result := new extra
  fi;
  if not (e=e0 and s is extra and s.top=none)
  then
    result.top := new link(e);
    result.top.prev := s.top
  fi;
```

```
end push;
unit equal : function(sp,sd :stack) : boolean;
  var s1, s2 : stack;
begin
  if ((sp is normal and sd is normal) or (sp is extra and sd is extra ))
  then (* the types of stacks sp and sd are conforming *)
    s1 := sp;
    s2 := sd;
    result := true;
    while(not s1.top=none and not s2.top=none and result)
    do
    (* till now both stacks are not empty and on the top one found equal elements *)
      result := top(s1).eq(top(s2));
      if result then
      s1 := pop(s1);
      s2 := pop(s2);
     fi;
    od;
    result := (result and s1.top = none and s2.top=none)
  else (* the types of the stacks sp and sd are different *)
    result := false;
  fi;
end equal;
begin
  e0 := new elem
end Stacks;
```

4. The Proof of Correctness

We present a proof of the correctness of the module Stacks with respect to the algebraic specification AxS. We shall prove that the operations (functions) of the class Stacks satisfy the specification (i.e. the formulas Ax1-Ax4).

In the sequel we shall use the following property for arbitrary type T, $[xx := new\ T](xx\ in\ T \wedge \neg xx = none)$, this is one of the axioms of the Loglan programming language.

The class Stacks shall be considered as a description of the algebraic system M

$$M = \langle E \cup S, top, pop, push, empty, equal, eq \rangle$$

where the carriers are: the set E of all objects of the type elem $E = \{e : e\ in\ elem\}$and the set S of all objects of the type stacks $S = \{s : s\ in\ stack\}$

Moreover, there are only two methods to create an element of the set S: either by creating a normal stack ($s := new\ normal$) or by creating an extra stack ($s := new\ extra$). One can say that new_normal and new_extra are two constants which belong to S.

From the definition of the class *stack* it follows that $Stacks \models s\ in\ stack \equiv (s\ in\ normal \cup s\ in\ extra)$ hence

$S = \{s : s\ in\ normal\} \cup \{s : s\ in\ extra\}$

The operations of M are: *top*, *pop*, *push*, *empty*, *equal* as defined by the functions of the class Stacks and *eq* a Boolean function which compares two elem objects.

Lemma 4.1. The algorithm in the body of the function *equal* always terminates i.e. the result of the function *equal* is always defined.

Proof:
The full proof is quite lengthy, it uses the similar arguments as the proof in [Sa 1978]. Let us sketch the arguments. First, observe that the class *link* is hidden, inaccessible from the outside of the class Stacks. Next, observe that the only reason for the eventual looping in the while instruction of the function *equal* would come from a manipulation on *prev* attribute of *link* objects. Only the *push* and *pop* functions update this attribute. One can analyze the assignments done there and prove that they keep an invariant:

the *link* objects referenced by a *top* attribute in a *stack* object always form a *finite list without cycles.*

This guarantees that *equal* operation will always terminate. □

Proposition 4.1. Let us remark that the programs defining the operations: *push*, *top*, *pop*, *empty* do not loop i.e. they always terminate.

Proposition 4.2. The operations *top* and *pop* terminate correctly i.e. without raising an exception iff the argument is not *empty*. in other words the domain of these operations is expressed by the formula $\neg empty(s)$.

Proposition 4.3. For every stacks *sp* and *sd*,
$equal(sp, sd) \Rightarrow top(sp).eq(top(sd))$
$equal(sp, sd) \Rightarrow equal(pop(sp), pop(sd))$
$top(sp).eq(top(sd) \wedge equal(pop(sp), pop(sd)) \Rightarrow equal(sp, sd)$

Proof:
The first and the second implication are the evident consequences of the definition of the function *equal*. For the proof of the third implication let us remark that the following implication is a tautology

$\gamma \Rightarrow (r := \mathbf{true}; \mathbf{while}\ \gamma \wedge r\ \mathbf{do}\ r := \beta;\ K\ \mathbf{od}\ \alpha \Leftrightarrow r := \beta;\ K;\ \mathbf{while}\ \gamma \wedge r\ \mathbf{do}\ r := \beta;\ K\ \mathbf{od}\ \alpha)$

It remains to observe that in our case the loop **while** is the body of the equal function, the formula β is the condition $top(sp).eq(top(sd))$, finally, the program K does $pop(sp)$ and $pop(sd)$ and the loop **while** now tests the equality of $pop(sp)$ and $pop(sd)$ stacks. □

In the sequel we shall use reflexivity, symmetry, transitivity and extensionality properties of *eq* and *equal* functions. As concerns the *eq* function we must assume that any application of Stacks will guarantee these properties.

The properties of *equal* are easily deducible from the definition of the *equal* function.

Proposition 4.4. The following formulas are valid in the structure of Stacks

$equal(s, s)$
$equal(s, s') \Rightarrow equal(s', s)$
$equal(s, s') \wedge equal(s', s'') \Rightarrow equal(s, s'')$
$e.eq(e') \wedge equal(s, s') \Rightarrow equal(push(e, s), push(e', s'))$
$equal(s, s') \Rightarrow equal(pop(s), pop(s'))$
$equal(s, s') \Rightarrow top(s).eq(top(s'))$

Let us consider the following formula (*)

$$(*) \quad (\forall e\ in\ elem)(\forall s\ in\ stack)$$
$$\left(\begin{array}{c} (empty(s) \Leftrightarrow (E\ result_E) \wedge (push(e, s) = Ph\ result_{Ph}) \wedge \\ (\neg empty(s) \Rightarrow top(s) = T\ result_T) \\ \wedge (\neg empty(s) \Rightarrow pop(s) = P\ result_P) \wedge (equal(s1, s2) \Leftrightarrow Eq\ result_{Eq}) \end{array} \right)$$

where E, Ph, T, P, Eq denote the bodies of the functions *empty*, *push*, *top*, *pop* and *equal*, respectively, and $result_E$, $result_{Ph}$, $result_T$, $result_P$, $result_{Eq}$ are the variables of the corresponding types.

From the lemma 1 and the remarks 1-2 it follows that the module Stacks is a model for the formula (*). Hence it suffices to prove that the properties Ax1-Ax4 are the consequences of the formula (*).

In the sequel we shall use the following notation in order to increase the readability $e =_E e'$ instead of $e.eq(e')$ and $s =_S s'$ instead of $equal(s, s')$

Proposition 4.5. $\models (\forall e\ in\ elem, \forall s\ in\ stack)\ \neg empty(push(e, s))$

Proof:
Let us consider separately the case of a normal stack and of an extra stack. Using the standard properties of the assignment and of the conditional instructions, we have from (*) the following properties of the push operation: $(s\ is\ normal\ \wedge\ s' = push(e, s)) \Rightarrow (s'\ is\ normal\ \wedge s'.top = none\ \wedge s'.top.el =_E e \wedge\ s'.top.prev = s.top)$ and $(s\ is\ extra\ \wedge s' = push(e, s)) \Rightarrow [(e = e0 \wedge\ s.top = none) \wedge s'\ is\ extra\ \wedge s' = s] \vee [\neg\ (e = e0\ \wedge s.top = none) \wedge s'.top.el =_E e \wedge s'.top.prev = s.top \wedge s'\ is\ extra]$. Hence $(s' = push(e, s) \Rightarrow ((s'\ is\ normal \wedge \neg s'.top = none) \vee s\ is\ extra)$. From the definition of the function empty we have $empty(s) \Leftrightarrow (s.top = none \wedge s\ is\ normal)$. Hence $(s' = push(e, s) \Rightarrow \neg empty(s')$ □

Proposition 4.6. $Stacks \models (\forall e\ in\ elem)\ top(push(e, s)) =_E e$

Proof:
Let $s' = push(e, s)$. From the definition of the function top we have $(s'\ in\ normal \wedge e' = top(s') \wedge \neg empty(s')) \Rightarrow (s'\ in\ normal \wedge e' =_E s'.top.el)$ and $(s'\ in\ extra \wedge e' = top(s')) \Rightarrow [s'.top = none \wedge e' =_E e0] \vee [\neg s'.top = none \wedge e' =_E s'.top.el]$. From the proposition 3 and by properties of $push$ operation we have $(s' = push(e, s) \wedge s\ in\ normal \wedge e' =_E top(s')) \Rightarrow (s'.top.el =_E e \wedge e' =_E s'.top.el)$ or $(s' = push(e, s) \wedge s\ in\ extra \wedge e' = top(s')) \Rightarrow (s'.top = none \wedge e' =_E e0) \vee (\neg s'.top = none \wedge e' =_E s'.top.el \wedge s'.top.el =_E e]$.This implies immediately $(s' = push(e, s) \wedge e' =_E top(s')) \Rightarrow e' =_E e$. Hence the axiom $top(push(e, s)) =_E e$ is valid in the structure Stacks. □

Proposition 4.7. Stacks$\models (\forall s\ in\ stack)(\neg empty(s) \Rightarrow push(top(s), pop(s) =_S s$

Proof:
By the definition of pop we have $(\neg empty(s) \wedge s' = pop(s)) \Rightarrow ([s'\ is\ extra\ \wedge s.top = none \wedge s'.top = none] \vee [s'\ is\ extra\ \wedge \neg s.top = none \wedge s'.top = s.top.prev])$. From the properties of $push$ and of top and from the above we can deduce three formulas: first $(\neg empty(s) \wedge e' = top(s) \wedge s' = pop(s) \wedge s\ in\ normal \wedge s'' = push(e', s')) \Rightarrow [e' = top(s) \wedge s'.top = s.prev \wedge s''\ in\ normal \wedge s''.top.el =_E e' \wedge s''.top.prev = s'.top]$ second $(e_1 = top(s) \wedge s' = pop(s) \wedge s\ in\ extra \wedge \neg s.top = none \wedge s_2 = push(e', s')) \Rightarrow [e_1 = top(s) \wedge s_1.top = s.prev \wedge s_2\ in\ extra \wedge s_2.top.el =_E e_1 \wedge s_2.top.prev = s'.top]$ third $(e_1 = top(s) \wedge s' = pop(s) \wedge s\ in\ extra \wedge s.top = none \wedge s_2 = push(e_1, s_1)) \Rightarrow (s_1\ is\ extra \wedge s.top = none \wedge s'.top = none)$.

All the tree formulas together imply $(\neg empty(s) \wedge e_1 = top(s) \wedge s_1 = pop(s) \wedge s_2 = push(e_1, s_1)) \Rightarrow [s_2.top.el =_E top(s) \wedge s_2.top.prev = s.prev]$ which means that the following formula is valid $(\neg empty(s) \wedge e_1 = top(s) \wedge s' = pop(s) \wedge s\ is\ normal \wedge s_2 = push(e_1, s_1)) \Rightarrow s_2 =_S s$. Hence axiom Ax 4 $(\neg empty(s) \Rightarrow push(top(s), pop(s)) =_S s$ is valid in the structure determined by the class Stacks. □

Proposition 4.8. $Stacks \models (\forall e\ in\ elem, \forall s\ in\ stack)\ pop(push(e, s)) =_S s$

Proof:
From the definition of the function pop we have: $(\neg empty(s) \wedge s' = pop(s)) \Rightarrow ([s'\ is\ normal \wedge s'.top = s.top.prev] \vee [s'\ is\ extra\ \wedge s.top = none \wedge s'.top = s.top.prev] \vee [s'\ is\ extra \wedge s.top = none \wedge s'.top = none])$ By the proposition 4 $s' = push(e, s) \wedge \neg empty(s')$ Thus using the

properties of the if_then_else construction we have for normal stacks $(s' = push(e,s) \wedge s \text{ is } normal \wedge s'' = pop(s')) \Rightarrow [s' \text{ is } normal \wedge s''.top = s'.top.prev]$ and for extra stacks $(s' = push(e,s) \wedge s \text{ is } extra \wedge s'' = pop(s')) \Rightarrow [s' \text{ is } extra \wedge \neg s'.top = none \wedge s''.top = s'.top.prev] \vee [s'' \text{ is } extra \wedge s'.top = none \wedge s''.top = none])$. By the properties of push operation we have $s' = push(e,s) \Rightarrow [\neg s'.top = none \wedge s'.top.prev = s.prev \wedge s'.top.el =_E e]$ Hence $(s' = push(e,s) \wedge s \text{ is } normal \wedge s'' = pop(s')) \Rightarrow [s' \text{ is } normal \wedge s''.top = s.top]$ and $(s' = push(e,s) \wedge s \text{ is } extra \wedge s' = pop(s')) \Rightarrow [s'' \text{ is } extra \wedge \neg s'.top = none \wedge s''.top = s.top]$. Which implies $(s' = push(e,s) \wedge s = pop(s'))s' =_S s$ that is, the axiom 3 $pop(push(e,s)) = s$ is valid in the Stacks structure. □

Let us resume the observed properties.

Theorem 4.1. *The algebraic specification of stacks AxS is valid in the class Stacks.*

Proposition 4.9. The algebra implemented by the class Stacks is a computation structure [AlgSpec] satisfying the (algebraic) axioms of stacks.

$$Stacks \in Gen(\Sigma, Ax1 - Ax4)$$

Σ is the signature of the structure of stacks.

5. The Pathology of the Model

The presented model is correct with respect to the given specification. Now we are going to observe that it has a big disadvantage of being pathological.

Proposition 5.1. There are objects of class stack for which one can *pop* without end, in other words there exists s in stacks such that the program

$$\textbf{while } \neg\ empty(s) \textbf{ do } s := pop(s) \textbf{ od}$$

never terminates.

Proposition 5.2. $Stacks \models \neg empty(newstack)$

In certain papers one adds this formula to the axioms of stacks. Newstack is supposed to be a constant. In absence of constants of composed type in our language we can use a variable and we forbid to change its value after its initialization. Let us define a variable newstack and let its value will be done by command *newstack* := *new norstos*. It is evident that $\neg empty(newstack)$ holds in the model defined by the class Stacks. We can add this formula as the fifth axiom Ax5 to the algebraic specification of stacks.

The following formula may be added too, $s \neq pop(s)$ as an additional sixth axiom to our specification. It will cause that our model will be rejected. But, it is quite straightforward to modify the model in such a way that it will satisfy all six axioms and still it will be non-standard.

We can give another, "mathematical" definition of the non-standard model described above.

Definition 5.1. Consider the following structure $\langle E \cup St, f_1, f_2, f_3, p_1, p_2 \rangle$

Let E be any set. As usual, E^*denotes the set of all finite sequences of elements of E. We define the set S (of stacks) as the union of two disjoint copies of E^*, $S = StN \cup StEx$. Let d denote a finite sequence of elements of E. The set StN is the set of all pairs $\langle d, 0 \rangle$, the set $StEx$ contains all the pairs $\langle d, 1 \rangle$. We define three operations and two predicates as follows:

for every $e \in E$,for every $s \in S$, $s = \langle d, i \rangle$ i is 0 or 1,

$$f_1(e,s) = \begin{cases} \langle e * d, i\rangle \text{ where } s = \langle d, i\rangle \text{ and } e * d \text{ is the concatenation of sequences } e \text{ and } d \\ s \quad \text{if } e = e0 \text{ and } s = \langle \emptyset,\ 1\rangle, \end{cases}$$

$$f_2(s) = \begin{cases} \langle d',\ i\rangle & \text{if } d \neq \emptyset \\ \langle \emptyset,\ 1\rangle & \text{otherwise,} \end{cases} \quad d' \text{ results from chopping off the first element of } d,$$

$$f_3(s) = \begin{cases} \text{the first element of the sequence } d \text{ if } d \neq \emptyset \\ e0 \quad \text{if the argument is } s = \langle \emptyset,\ 1\rangle \end{cases}$$

$p_1(s) \Leftrightarrow s = \langle \emptyset,\ 0\rangle$,
$p_2(s1,\ s2) =$ the identity predicate in S.

If one interprets *push* as f_1, *pop* as f_2, *top* as f_3, *empty* as p_1, *equal* as p_2 and *eq* as the identity in E then again we have a model of the axioms AxS and we observe that:

Proposition 5.3. The model described above and the model defined in section 2 are isomorphic .

6. Conclusions

We have constructed the module Stacks that satisfies all the properties mentioned in the specification and which can not be used in any application. In the view of our example we can not share the opinion that "it is appropriate for program verification and development to regard the class $Gen(\Sigma, E)$ of all Σ-structures satisfying the axioms E as the semantics of the specification (Σ, E) " [AlgSpec]. For the class is *too large* and admits the models which are to be avoided in practice in spite of being computable i.e. programmable .

If the meaning of the specification (Σ, E) is the initial algebra in the class $Gen(\Sigma, E)$ then other problems arise:

- Our model being not the initial algebra in the class $Gen(\Sigma, E)$ should be rejected. What would be a criterion enabling to reject our model? Let us remark that the programmers may need a criterion adequate to the programming language they use. How to explain a programmer that the models like ours should be rejected? on which formal base? The preference should be given to syntactic criterions above semantic ones.
- One would expect that the additional part of the specification which states " of all models of axioms choose the initial one" will find its syntactical counterpart and that
- it will enable to *add more properties* to the set E of axioms.

We propose to study these questions. Some results will appear in [PAS]

6.1. Reminder

We would like to remind that it is possible to create algorithmic specifications by adding algorithmic properties to the algebraic ones . Many problems find simpler solutions then.

The algorithmic specification of stacks consists of the axioms of algebraic specification and moreover of the following formula

$(\forall s)$ **while** $\neg empty(s)$ **do** $s := pop(s)$ **od true**[3]

We were able [AL, ATS] to prove the following facts:

1. (IDENTIFICATION) any model of the axioms is isomorphic to a standard model, hence the identification problem is solved in this case correctly,

[3]The meaning of this algorithmic [AL]formula read as follow: for every $s \in S$ the program *while* $\neg empty(s)$ *do* $s := pop(s)$ *od* always terminates. It is the negation of proposition 5.1.

2. (COMPLETENESS) the set Val of all formulas valid in all models of algorithmic specification of stacks and the set Th of all formulas provable from the algorithmic specification of stacks are equal, $Val = Th$

   this is an immediate consequence of the completeness property of algorithmic logic.

From 1 and 2 follow:

3. (CRITERION) by 1 we know that any implementing module which satisfies the axioms is correct.

3a (Proofs of correctness of implementations) At various occasions we gave proofs of implementing module showing that algorithmic definitions occurring in the implementing module imply the axioms of a specification [AL, ATS]. It means that the algorithmic specifications are good criterions for acceptance or rejection of a module. Moreover, there is a close relationship between implementing modules and the algorithmic specifications which makes the task of proving easier and more natural.

4. (PROVABILITY of *applications* i.e. of program's properties) Suppose we have an application (an algorithm) P and a property α of the algorithm, for example: termination of P or correctness of P w.r.t. certain pre-condition and post-condition

   the property α is valid in any correct implementation of the algorithmic specification of stacks
   if and only if α is provable from the axioms of the specification.

For algebraic specification the point 4 is unreachable for two reasons:

- the language used in algebraic specifications does not admit the programs nor the properties of programs, hence we are not able to express the semantic properties of programs,
- an eventual proof of, say, termination property, uses the information not only outside the syntax but also outside the semantics of axioms of algebraic specification. It will be necessary to use the knowledge on initial (resp. terminal algebras ...) But where it comes from?

Remark 6.1. As a side effect of the present considerations, the reader may remark the close relationship between specifications and classes in object oriented style of programming[AL, Loglan].

References

[AL] Mirkowska G., Salwicki A., *Algorithmic Logic*, PWN and Reidel Pub. Co., Warszawa, Dordrecht 1987, pp.1-368

[AlgSpec] Wirsing M., *Algebraic Specification*, in Handbook of Theoretical Computer Science, vol. B, ed. J. van Leeuwen, Elsevier Sci. Pub. 1990, pp.675-788

[ATS] Salwicki A;, *On algorithmic theory of stacks*, in Procedings of MFCS'78, Zakopane (J;Winkowski ed.), Lecture Notes in Computer Science vol. 64, pp 452-461, full paper in Fundamenta Informaticae 3 (1980), pp.311-332

[DTS] Thatcher J.W., E.G. Wagner and J.B. Wright, *Data type specification: parametrization and the power of specification techniques*, ACM TOPLAS 4 (1982), pp.711-773

[Hoare] Hoare C.A.R., *Proof of correctness of data representation*, Acta informatica, 1972, pp.271-281

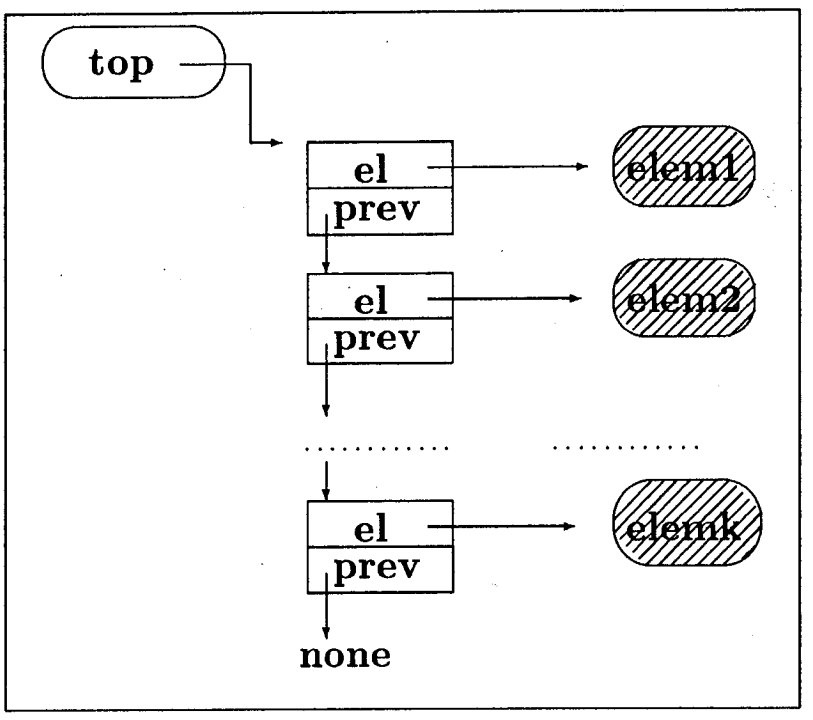
[Loglan] Bartol, W.M. et al.*Report on Loglan programming language,* PWN, Warszawa, 1983, pp.1-143 *see also* http://www.univ-pau.fr/~salwicki/loghome.html

[PAS] Mirkowska, Salwicki, Srebrny, Tarlecki, *Programmable Algebraic Specfications...*to appear

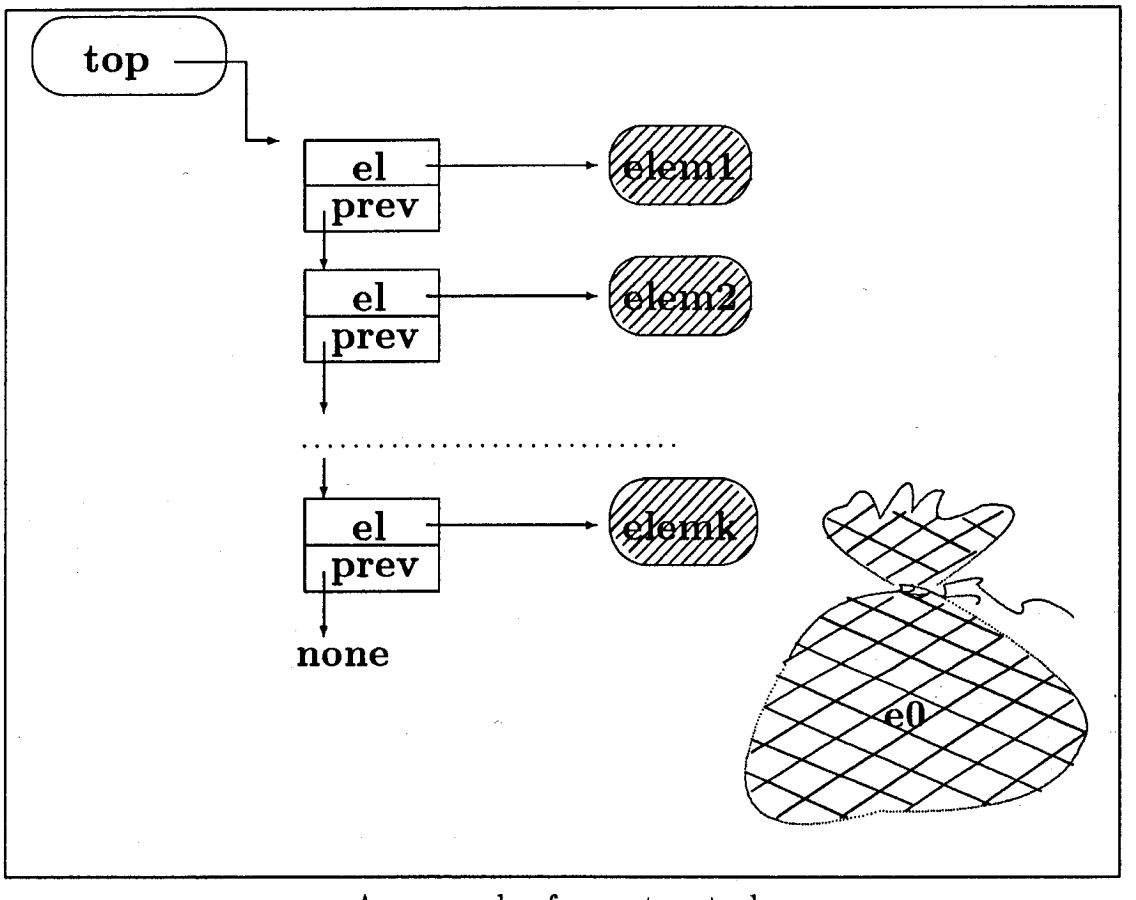
[Ten] Tennenbaum, S., *Non-Archimedean...*, Notices of AMS, 1959

7. Appendix

Two examples of stacks follow



An example of an normal-stack



An example of an extra-stack