6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

On the correctness of object classes

Péter Szlávi

Informatics Methodology Group,
University Eötvös Loránd
e-mail: szlavi@ludens.elte.hu

Abstract

The cardinal purpose of teaching programming is to demonstrate a method of developing correct programs. Reliability and correctness are especially important in cases of programs that are intended for multiple reuse. The most characteristic examples of such program-tools are the ones belonging to data types (or data type classes). The method we outline for program development assists in producing correct programs with mathematical formalism.

The main point of this method is that we declare statements that refer to the examination of the abstract type class's operations formulated with algorism. Since there are well-known methods for proving these statements, we do not have to deal with it in this paper.

We are going to apply a single example throughout the article. The theorems we need to support our steps are to be declared, but our methods generally do not require elaboration of their proof.

Categories and Subject Descriptors: D.1.4 [Programming Techniques]: Sequential Programming; D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.1 [Software Engineering]: Requirements/ Specification; D.2.4 [Software Engineering]: Software/Program Verification – correctness proof; D.2.10 [Software Engineering]: Design – methodology; D.2.11 [Software Engineering]: Software Architectures – data abstraction

Key Words and Phrases: Program methodology, data abstraction, data type class, proof of program correctness

1. The method from a bird's eye view

The data type class we want to produce is to be called target type. We describe this data type class with algebraic formalism: the parameters used to define a certain type, the operations used to manage the data of this type, as well as the way they work and what they yield. In short: we define the abstract syntax and semantics of the operations belonging to this type. Semantics is characterized

by an algebraic system, which is the **abstract specification** of the target type ($\text{SPEC}_a^{(ax)}$).

Problem 1: *How can the completeness or consistency of an algebraic system be proved?*

We choose a suitable type on which we will build the one we study. This type is specified algebraically too. Let us call it import type. This way, we get to the **import specification**. ($\text{SPEC}_{imp}^{(ax)}$)

In the next step we formulate the representation of the target type with the assistance of the import type. Here we define a **mapping** between the representation and the range of the target type. (φ)

Problem 2: *What property do we expect the mapping to have?*

The equations set up in the abstract specification have to be true with the representation as well. By this, we have given the **first concrete specification** of type. ($\text{SPEC}_c^{(ax)}$)

Using the algebraic description, we formulate expectations for each operation: we give their pre- and post-conditions. This way, we get to the **second abstract specification** ($\text{SPEC}_a^{(pp)}$ where *pp* refers to the **p**re- and **p**ost-condition.)

Problem 3: *What guarantees that the pre- and post-conditions are in harmony with axioms?*

We enumerate the operations, and in a "standard" algorithmic language we prepare the algorithms of the procedures and functions realizing them.[1] Now, we have achieved our purpose. The final realization of the type is done. We complete our procedures and functions with the pre- and post-conditions that are based on their own abstract pairs. This is the **second concrete specification**. ($\text{SPEC}_c^{(pp)}$)

Problem 4: *Do the bodies of the procedures and functions meet the expectations? Can it be proved that when the precondition realizes, the post-condition is still true after the transfor mation?*

On figure 1 below we summarize the problems to be solved.

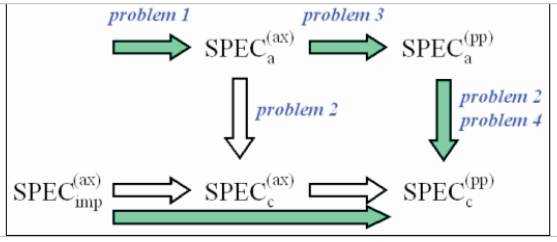

Figure 1 – The steps and the problems of the method

The coloured arrows illustrate the way we chose to follow to present the method.

[1]The word "standard" refers to a supposition that the semantics of the instructions is unambiguous and well known; therefore we do not specify them.

2. Indispensable concepts

As is seen, we gradually achieve our purpose, forming mapped specification into specification.[2] This mapping has to guarantee that the mapped type is equal to the original one. Actually, the formal part of our investigation is focussed exactly on this.

Apart from the temporary conditions of the specifications, we can declare that two specifications play leading roles in our deduction: the abstract specification and concrete (final) one. Because of the formal deduction, we add two further specifications: one belonging to the parameter and one to the import. The figure below illustrates the relations of these specifications.

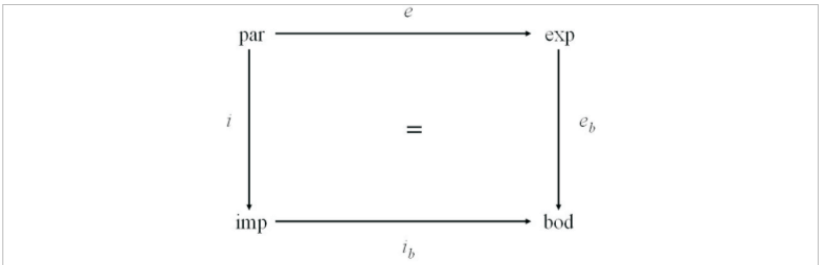

Figure 2 – A formal definition for type class can be given by the morphism diagram

We write *exp* and *bod* to denote abstract (original) and concrete (final) specification, respectively. The *par* specification formulates the expectations of the parameter, while the *imp,* as the basis of the representation, denotes the requirement of the import. The mappings e, i, e_b and i_b are morphisms, that is, they hold all the properties of their own operations, essentially. [3]

In order to avoid repeated noting, we mention only the "extras" of the current specification. For example, some characteristics that appear in the *par* do not reappear formally in the *exp*. This practice yields a great profit in the case of the *bod*, because there we imply everything described about *imp* and *exp* without rewriting them.

Consequently, the type is defined by a so-called double specification. Henceforth, this concept requires a formal approach, which is provided below.

This specification of the type class is a **double specification**, which contains

- an abstract specification: SPEC_a and
- a concrete specification: SPEC_c.

In details:

[2]The definition of the specification can be found in [1/11 pp.,2/24 pp.].
[3]The precise definition of the morphism can be read in [1/27 pp., 2/25,32 pp.].

$$
\begin{aligned}
SPEC_a &= (A, F, E_a) \\
A &= \{A_0, A_1, \ldots, A_n\} \\
F &= \{f_0 \to A_0, \ldots, f_m{:}A_j \ldots A_k \to A_l\};
\end{aligned}
$$

$$
\begin{aligned}
SPEC_c &= (C, G, E_c) \\
C &= \{C_0, C_1, \ldots, C_n\} \\
G &= \{g_0 \to C_0, \ldots, g_m{:}C_j \ldots C_k \to C_l\}
\end{aligned}
$$

where A and C include the sets composing the two specifications; while F and G are the corresponding operation sets. We paired the operations of the types: the pair of f_0 in A is g_0 in C etc. We assume that the members of the pairs have identical signature.

Below we define the "harmony," the most important feature characterizing the nexus of these specifications.

Definition.

Let $d_a = (A, F, E_a)$ *be an abstract specification, and*
$d_c = (C, G, E_c)$ *be a concrete specification with common signature.*
Let $\varphi\colon C \to A$ *be a morphism.*

If

(1) *C is a representation of A with morphism* φ.

(2) $(\forall f_i \in F)(c \in C \wedge \varphi(c) \in A \wedge f_i(\varphi(c))$ *is definite* $\Rightarrow g_i(c)$ *is definite too.*

(3) $(\forall f_i \in F)(c \in C \wedge c' = g_i(c) \wedge c' \in C \wedge \varphi(c) \in A \wedge \varphi(c') \in A \Rightarrow$
$f_i(\varphi(c)) = \varphi(g_i(c))$.

then d_c *is* ***correct*** *with respect to* d_a.

A remark to the (1): The morphism $\varphi\colon C \to A$ is given. The C is the representation of A if $\forall a \in A\colon \exists c \in C\colon a = \varphi(c)$.

Theorem (correct representation).

Given an algebraic specification $\mathrm{SPEC}_a = (\Sigma_a, \mathrm{E}_a)$ of an abstract data type d_a=(A,F), a specification $\mathrm{SPEC}_c = (\Sigma_c, \mathrm{E}_c)$ of a concrete data type d_c=(C,G), and a morphism $\varphi\colon C \to A$. Let be $\mathrm{F}_c \subset F$ a set of constructors. If for $\forall f_c \in F_c$ holds that

$$\mathrm{a} \in A \wedge f_c(\mathrm{a}) \in A \wedge g_c(\mathrm{c}) \in C \wedge \mathrm{a} = \varphi(c)$$

and

$$\forall \mathrm{c} \in C \wedge \forall f_c \in F_c\colon f_c\varphi(\mathrm{c})) \in \varphi(g_c(\mathrm{c})),$$

then

C is a ***representation*** of A.

The next theorem in the focus of our method gives controllable and sufficient conditions for the correctness of the concrete specification, with respect to the abstract one. It contains a few concepts that are worth mentioning:

- type invariant – an assertion that holds for every element in the base set of a type (I_a, I_c);
- precondition – it is a predicate that describes the states in which the program may be started (pre_f, pre_g);
- post-condition – describes the states after program termination ($post_f$, $post_g$).

Theorem(correctness of the concrete specification) **(ThCCS)**

Given

$d_a = (A, F, E_a)$; $\{pre_{fi}(a)\}\ a' = f_i(a)\ \{post_{fi}(a,a')\} \in E_a$, $f_i \in F$, $i=0,\ldots,m$; and

$d_c = (C, G, E_c)$; $Q_{gi} \in E_c$, $g_i \in G$, $i=0,\ldots,m$; here procedure Q_{gi} for calculating the value of the operation g_i.

Assume d_a and d_c are specifications with respect to common signature.

$A = \{ a \mid I_a(a)\}$,

$\{\text{"true"}\}\ a = f_0\ \{post_{f0}(a)\}$,

$\{pre_{fi}(a)\}\ a' = f_i(a)\ \{post_{fi}(a,a')\}$, $i=1,\ldots,m$;

$C = \{ c \mid I_c(c)\}$,

procedure g_0 begin Q_0 end;

procedure g_i begin Q_i end; $i=1,\ldots,m$;

and

$\varphi : C \to A$ be a morphism.

If the following theories are verified:

(i) $(\forall c \in C)(\ I_c(c) \Rightarrow I_a(\varphi(c))$;

(ii) $\{\text{"true"}\}\ Q_0\ \{post_{f0}(\varphi(c)) \wedge I_c(c)\}$;

(iii) $(\forall f \in F) : \{pre_{fi}(\varphi(c\ )) \wedge I_c(c\ )\}\ Q_i\ \{post_{fi}(\varphi(c\ ),\varphi(c')) \wedge I_c(c')\}$;

where (ii) and (iii) are theories of total correctness,

then the concrete specification d_c is correct with respect to the abstract specification d_a.

3. Observing the method

The particular type the specification of which we examine is the well-known *bag*. The *bag* is the generalization of the *set*. In the *bag*, unlike in the *set*, more than one piece of the item can be collected.

The specification of the type class bag could be as follows:

bag(data, n : natbool) **is a class specification** =	
parameters = **natbool** +	
sorts : data	
oprs : _ = _ : data data → bool	
eqns : a,b,c∈data	
($data_1$)	a = b ⇒ b = a
($data_2$)	a = b ∧ b = c ⇒ a = c

The *bag* is a composite data structure that collects items of any (identical) type, but its cardinality is finite. So it has some parameters:

1. the element type – *data*
2. the maximum number of the members – *n: natbool*

The first line of the specification shows this.

It is necessary to specify what we expect from the type of the parameters, since they meddle in the export and import specifications, as you have seen on fig. 2. The part *parameters* define the *par* parameter specification. Since we have previously given the formal specification of the type *natbool*, we do not write it here. Furthermore, our expectation is that there is an operation "equal" of the type data with usual properties ($data_1$=symmetry, $data_2$=transitivity). We give the abstract syntax of the *data* operations after *oprs* and the axioms after *eqns*.

exports =	
class sorts : bag	
oprs :	bempty : → bag
	binsert : bag data → bag
	bdelete : bag data → bag
	many : bag data → natbool
	bsize : bag → natbool
eqns :	b∈bag; a,e∈data
(bag_1)	bdelete(bempty,a) = bempty
(bag_2)	many(bempty,a) = zerus
(bag_3)	bsize(bempty) = zerus
(bag_4)	bdelete(binsert(b,a),e) = **if** a=e **then** b **else** binsert(bdelete(b,e)) **fi**
(bag_5)	many(binsert(b,a),e) = **if** a=e **then** succ(many(b)) **else** many(b,e) **fi**
(bag_6)	bsize(binsert(b,a)) = succ(bsize(b))
(bag_7)	bsize(binsert(b,a)) > n ⇒ binsert(b,a) = "undefined"

The purpose of the part *export* is to clarify what we demand of the type class (the *bag*). This, we can say, is the **abstract** notion of the *bag,* since it has no connection with the later representation of the type. Also, it is a type specification with similar structure (as far as the parts *sort*, *oprs* and *eqns are concerned*).

The algebraic system needs to be complete and consistent, which can be achieved easily, as the next proposition states:

Proposition (guarantee of the completeness and consistency of the axiom system) **(PrGCC)**

For each object of the type, created by constructors (or multiple calling of constructors), we give the values of any other (non-constructor) operators.

In such a way, the axioms are built. Now we have received the answer to problem 1 in the previous chapter, but still, the completeness and consistency cannot be proven theoretically. The numerous practical cases, however, encourage us to believe in the correctness of the proposition above.

Similarly, we define the *bag*: the operators *bempty* and *binsert* are constructors by which any bag objects can be generated. Therefore, we group any other opera-

tors around them. (*bempty* → bag_1-bag_3, *binsert*→ bag_4-bag_7) Some notes about each axiom are as follows:

- ad bag_1: the operator *bempty* does not change the empty *bag*
- ad bag_2: the number (i.e. the multiplicity) of the items in the empty *bag* is 0
- ad bag_3: the size of the *empty bag* is 0
- ad bag_4: if the data arguments of the operators *bdelete* and *binsert* are not identical, the order of the operators is invertible; otherwise the operator-pair "*binsert* and then *bdelete*" has no effect on the *bag*
- ad bag_5: the operator *binsert* increases the number of the given item by one
- ad bag_6: the operator *binsert* increases the size of the bag by one as well
- ad bag_7: it is impossible to exceed the maximum size of the *bag* by putting more items in it; if we do try it, the *bag* gets into an undefined condition.

imports =	
sorts :	vector
oprs :	nil : → vector
	_ [_] ::= _ : vector data natbool → vector
	_ [_] : vector data → natbool
eqns :	v∈vector; i,j∈data; e∈natbool
(vect_1)	nil[i] = zerus
(vect_2)	(v[i]::= e)[j] = **if** i=j **then** e **else** v[j] **fi**;

Now we trace the representation of the *bag* back to the *vector*.

In the *import* specification we introduce the *vector* on which the representation and the implementation of the *bag* will be based, by describing the abstract syntax and semantics of its operations. In order to formulate the vector, we use the *data* parameter type as its index type. The elements of the vector will be the numbers from *natbool*. It is clear that member *e* of the *bag* is the index of the *vector*, and its multiplicity is an item of the *vector*, as well.

Axiom $vect_1$ expresses that the *vector* is formed with well-defined value (zerus).

body =	
oprs :	rep: vector natbool → bag
	_+1 = succ(_)
	_-1 = prec(_)
	0 = zerus
eqns :	v∈vector; e∈data
(bod_1)	bempty = (nil, 0)
(bod_2)	binsert((v,m),e) = (v[e] :: = v[e] + 1, m+1)
End bag;	

The *body* is the fourth specification. Its role is to connect the **concrete** representation with the **abstract** target type, i.e. the *bag*. This connection is set up by the so-called representation function φ.

First, we give the syntax of this mapping. From this, it is clear that the representation of the *bag* consists of the pair of a *vector* and a *number*.

The formalism can be simplified by applying morphism between the specifications. We change the names of the three frequently used operations of *natbool*, precisely *zerus*, *succ* and *pred*, to their traditional equivalent.

According to the meaning of the morphism diagram, everything we have defined so far, will map here. This morphism, however, is a mapping that includes representation too. This is why we have to specify the representation here as well.

Referring to the constructivity of the type *bag,* we give the φ–images of the two constructors, *bempty* and *binsert*, which can generate all the members of the type set.

The bod_1 asserts that the operation *bempty* produces a value-pair of an empty *vector* and a number *0* indicating the size of the *bag,* as the representation of *bag*: $bempty_a = \varphi(bempty_c)$. The bod_2 states that an extra element put into the *bag,* increases the multiplicity of the suitable member of the *vector* and the size of the *bag*: $binsert_a(\varphi(c),e) = \varphi(binsert_c(c,e))$. In the formulae, the subscripts are used to indicate where the operation belongs: op_a to the abstract specification and op_c to the concrete one. The morphism between the specification SPEC_{exp} and SPEC_{body} is so-called containning and representation-morphism [2/49], so now we do not need other axioms in the *body*, we can omit them.

We have reached the answer to problem 2. What is φ like?

The properties of the mapping φ **(Prφ)**

1. every concrete object should have an image in the φ-range,
2. it should be so-called "operation-preserving" (i.e. $\varphi(op_c(\mathrm{x,y,}\ldots)) = op_a(\varphi(\mathrm{x}),\varphi(\mathrm{y}),..)$ for all x, y, ... in C, and for all op_a and op_c defined on A and C, respectively), and
3. every abstract object should have an "ancestor" in the φ-domain.

Properties 1 and 2 are together called "morphic," which, combined with the third one, satisfy the expectations of the representation mapping,

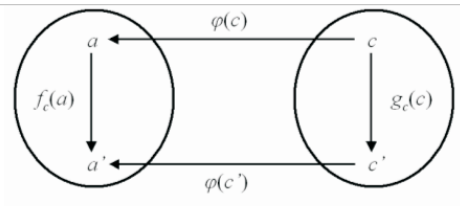

Figure 3 – The illustration of φ by the morphism diagram

In part *body* each concrete operator, including φ, is precisely defined. But we intend to describe them with their algorithms, as well. We would like to write a program, but there it is easy to make mistakes.

In the following, we are going to take steps towards proving correctness. Roughly, it means the following:

I. to give the explicit definition of the representation function, because it plays a significant role in proving, then

II. to implement the operations, in addition to the concrete representation, and prove their correctness with respect to the abstract specification.

I. Give a recursive definition to the representation function.

(φ_{sign})	φ : vector(data,natbool) $\times$ natbool $\rightarrow$ bag(data,natbool)
(φ_1)	$m = 0 \Rightarrow \varphi(v,m) =$ bempty
(φ_2)	$(\exists e \in data)(v[e] \geq 1) \Rightarrow \varphi(v,m) =$ binsert(φ(v[e]::= v[e]-1,m-1);e)

Now we rewrite the function in an explicit form, which was defined implicitly in part *body*. This is a plain transformation, which is a conventional form of functions.

It is worth considering the recursive definition. The second component of the concrete type set is to present the size of the *bag*, which, if it is 0, indicates the same state achieved by a create-like operation.

If the *bag* is not empty, there must be an element of type *data* that has been put into the *bag,* at least once; in other words, the value of the adequate element of the *vector,* representing the *bag,* is at least 1. This way, we trace the value of φ back to the previous state of the execution of the operation *binsert* referring to the element. This is the moment when φ, as expected, holds the property "morphic."

Three conditions of theorem **ThCCS** is to be proved.

Let us introduce the invariants to be used in the proving.

$$I_a(b) = 0 \leq bsize(b) \leq n$$
$$I_c(c) = 0 \leq m \leq n, \text{ ahol } c=(v,m)$$

Further on, the next lemma [1/56 pp.] plays a key role.

Lemma(conversion)

An algebraic specification can be converted to the form p/p of type, with the semantics unchanged.

Consequently, a transformation rule can be formulated for the creation of pre- and post-conditions. This is how we receive the answer to problem 4.

Transformation rules: **(TR)**

Give

$$E_a = \{\ldots, \alpha(a) \Rightarrow f_s(f_c(a)) = h(a), \ldots, \neg I_a(f_c(a)) \Rightarrow f_c(a) = \text{"undefined"}\},$$

where

$$I_a(a) = 0 \leq attr(a) \leq n.$$

Conversion.

1. for $\forall f_s \in F_s$: $\{\alpha(a) \wedge b=f_c(a)\}$ $b'=f_s(b)$, $\{b'=h(a)\}$
2. for $f_0 \in F_c$: {"true"} $a=f_0$ $\{I_a(a) \wedge a=f_0\}$
3. for $\forall f_c \in F_c \backslash \{f_0\}$: $\{I_a(f_c(a))\}$ $b=f_c(a)$ $\{ I_a(b) \wedge b=f_c(a)\}$

Hence it is clear that the condition i. of **ThCCS** holds, i.e. the next theorem can be proved:

Theorem(i-condition) **(ThiC)**

$(\forall c \in C)(I_c(c) \Rightarrow I_a(\varphi(c))$

Proof:

$I_c(v,m) = 0 \le m \le n \Leftrightarrow I_c(v,0) \vee I_c(v,m) \wedge 1 \le m \le n$

Let us inspect the truthfulness of the first part ($I_c(v,0)$):

$\varphi_1 \Rightarrow$ $\varphi(v,0)$=bempty $bag_3 \Rightarrow$ bsize(bempty)=0	} $\Rightarrow$	$0 \le$ bsize(bempty)$\le n$
	$\Rightarrow$	I_a(bempty)=$I_a(\varphi(v,0))$

The second part ($I_c(v,m) \wedge 1 \le m \le n$) can be verified by induction. Supposing that this is true for some m ($1 \le m < n$), and let us examine the case for m+1:

induction hypothesis$\Rightarrow$ $(\exists e \in$data$)$ $(v[e] \ge 1) \Rightarrow$

$\varphi_2 \Rightarrow$ $\varphi(v,m+1)$=binsert($\varphi(v[e]::=v[e]\text{-}1,m)$;e)

$bag_6 \Rightarrow$ bsize(binsert($\varphi(v[e]::=v[e]\text{-}1,m)$;e))=bsize($\varphi(v[e]::=v[e]\text{-}1,m)$+1

induction hypothesis$\Rightarrow$ $0 \le$ bsize($\varphi(v[e]::=v[e]\text{-}1,m) < n \Rightarrow$

$\Rightarrow 0 \le$ bsize($\varphi(v[e]::=v[e]\text{-}1,m)+1 \le n$

$\Rightarrow 0 \le$ bsize(binsert($\varphi(v[e]::=v[e]\text{-}1,m)$;e)) $\le n$

$\Leftrightarrow I_a(\varphi(v,m+1))$

Qed.

The condition (ii) of the theorem is true as well.

Theorem(ii-condition) **(ThiiC)**

{"true"} Q_0 $\{post_{f0}(\varphi(c)) \wedge I_c(c)\}$

Proof:

Q_{bempty}: **begin** m' $\leftarrow$ 0 **end;**

Let us add the pre- and post-condition generated by **ThCCS** to code.

{"true"} **begin** m' $\leftarrow$ 0 **end** $\{0 \le$ bsize(b)$\le n \wedge$ b=bempty$\}$

Now the correctness over the representation has remained to be proved, i.e.

{"true"} **begin** m' $\leftarrow$ 0 **end** $\{0 \le$ bsize(φ(v',m'))$\le n \wedge \varphi$(v',m')=bempty$\}$

Let us verify it.

The steps of transforming the post-condition:

$bod_1 \Rightarrow$ $\{0 \le$ bsize(φ(v',m'))$\le n \wedge$ (v',m')=(nil,0)$\} \Rightarrow$

$\{0 \le$ bsize(φ(nil,0))$\le n\}$

$\varphi_1 \Rightarrow$ $\{0 \le$ bsize(bempty)$\le n\}$

$bag_3 \Rightarrow$ $\{0 \le 0 \le n\} \Leftrightarrow$ "true"

Qed.

After all our duty is to show that the condition (iii) of **ThCCS** holds/is true, i.e.

$$(\forall f \in F \setminus \{f_0\}) : \{pre_{fi}(\varphi(c\ )) \wedge I_c(c\ )\}\ Q_i\ \{post_{fi}(\varphi(c\ ),\varphi(c')) \wedge I_c(c')\}.$$

The case of the concrete bag, the proof is to be performed for the operation *bdeletete*, *binsert*, *many*, and *bsize*. We will take the next steps.

a) we collect the bag's axioms with respect to the given operation,
b) using **ThC** we prepare the pre- and post-condition of the operation,
c) we assemble the implementation,
d) we state the proposition referring to the operation
e) and finally, we prove it; more exactly, in this paper we still do not prove it since this is well-known in programming.

II. Give algorithms for implementing concrete operations with correctness theorems.

bdelete:

a) Axioms:

(bag1)	bdelete(bempty,a) = bempty
(bag4)	bdelete(binsert(b,a),e) = **if** a=e **then** b **else** binsert(bdelete(b,e)) **fi**

b) In state oriented form we have:

Here we use the first case of **TR**, and combine the pre- and post-condition coming from the axioms bag_1, bag_4:

{y = bempty ∨ y = binsert(b,a)}
z ← bdelete(y,e)
{(y = bempty ∧ z = bempty) ∨ (y = binsert(b,a) ∧ e = a ∧ z = b) ∨
(y = binsert(b,a) ∧ e ≠ a ∧ z = binsert(bdelete(b,e),a)}

c) $Q_{bdelete}$(v,m,e):

begin
(m',j) ← (m,0);
while j ≤ m' **do** (j,v'[e]) ← (j+1,v[e]) **od;**
if v'[e] ≥ 1 **then** (v'[e],m') ←(v[e]-1,m-1) **fi;**
end;

d) Theorem to prove **(Th-bdelete)**

{(φ(v,m) = bempty ∨ φ(v,m) = binsert(b,a)) ∧ 0 ≤ bsize(φ(v,m))≤ n }
begin
(m',j) ← (m,0);
while j ≤ m' **do** (j,v'[e]) ← (j+1,v[e]) **od;**
if v'[e] > 1 **then** (v'[e],m') ←(v[e]-1,m-1) **fi;**
if v'[e] =1 **then** (v'[e],m') ← (0, m-1) **fi**;

end;
{((φ(v,m) = bempty ∧ φ(v',m') = bempty) ∨ (φ(v,m) = binsert(b, a) ∧ e=a ∧ φ(v',m') = b) ∨ (φ(v,m) = binsert(b, a) ∧ e≠a ∧ φ(v',m') = binsert(bdelete (b,e),a)) ∧ 0 ≤ bsize(φ(v',m'))≤ n }

binsert:

a) Axioms:

(bag_6)	bsize(binsert(b,a)) = succ(bsize(b))
(bag_7)	bsize(binsert(b,a)) > n ⇒ binsert(b,a) = "undefined"

b) In state oriented form we have:
bag_{67}⇒ bsize(binsert(b,a))≤ n ⇒ bsize(binsert(b,a)) = succ(bsize(b))
So we need to modify the third case of **TR** with adding above condition, that we got from bag_6 and bag_7.
{0 < bsize(binsert(b,a))≤ n ∧ 0 ≤ bsize(b)≤ n }
b' ← binsert(b,a)
{((0 < bsize(b')≤ n) ∧ b' = binsert(b,a) }
c) $Q_{binsert}$(v,m,e):
begin
j ← 0;
while j ≤ m **do** (j,v'[e]) ← (j+1,v[e]) **od;**
(v'[e], m') ←(v[e]+1,m+1)
end;
d) Theorem to prove **(Th-bdelete)**
{0 < bsize(binsert(φ(v,m), e))≤ n ∧ 0 ≤ bsize(φ(v,m))≤ n }
begin
j ← 0;
while j ≤ m **do** (j,v'[e]) ← (j+1,v[e]) **od;**
(v'[e], m') ←(v[e]+1,m+1)
end;
{(0 < bsize(φ(v',m'))≤ n) ∧ (φ(v',m') = binsert(φ(v,m), e) }

many:

a) Axioms:

(bag_2)	many(bempty,a) = zerus
(bag_5)	many(binsert(b,a),e) = **if** a=e **then** succ(many(b)) **else** many(b,e) **fi**

b) In state oriented form we have:
{b = bempty ∨ b = binsert(y, a) }
x ← many(b)
{(b = bempty ∧ x = zerus) ∨
(b = binsert(y, a) ∧ a = e ∧ x = succ(many(y))) ∨
(b = binsert(y, a) ∧ a ≠ e ∧ x = many(y))}
c) Q_{many}(v,m,e):
begin

x ← v[e]
end;
d) Theorem to prove **(Th-many)**
{φ(v,m) = bempty ∨ φ(v,m) = binsert(y, a) ∧ 0 ≤ bsize(φ(v,m))≤ n }
begin
x ← v[e]
end;
{(φ(v,m) = bempty ∧ x = zerus) ∨
(φ(v,m) = binsert(y, a) ∧ a = e ∧ x = many(y))+1 ∨
(φ(v,m) = binsert(y, a) ∧ a ≠ e ∧ x = many(y))}

bsize:

a) Axioms:

(bag3)	bsize(bempty) = zerus
(bag6)	bsize(binsert(b,a)) = succ(bsize(b))

b) In state oriented form we have:
{b = bempty ∨ b = binsert(y, a) }
x ← bsize(b)
{(b = bempty ∧ x = zerus) ∨
(b = binsert(y, a) ∧ x = succ(bsize(y)))}
c) Q_{size}(v,m):
begin
x ← m
end;
d) Theorem to prove qtextbf(Th-size)
{φ(v,m) = bempty ∨ φ(v,m) = binsert(y, a) ∧ 0 ≤ bsize(φ(v,m))≤ n }
begin
x ← m
end;
{(φ(v,m) = bempty ∧ x = 0) ∨
(φ(v,m) = binsert(y, a) ∧ x = bsize(y)+1)}

4. Summary

In this paper we have outlined a method for the appropriate deduction of the type class. To prove its correctness, we have used mathematical formalism, and we have based the derivation on numerous theorems. Our starting point is the algebraic description of the type: $SPEC_a^{(ax)}$.
Our final purpose, on the other hand, is to give the algorithmic specification of the type, $SPEC_c^{(pp)}$, which lies on a fixed representation of $SPEC_{imp}^{(ax)}$. The operations in the final specification are given by their algorithms. But to prove their correctness, we have to know the pre- and post-conditions of the operations. To generate this assertion-pair, we go back to the first specification $SPEC_a^{(ax)}$.

We have collected the problems emerging from our method, then we have presented some solutions for them The theorems used during our derivation, as summarized in Fig. 4, serve as theoretical and practical bases as well.

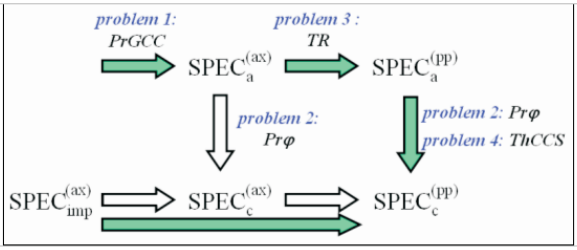

Figure 4 – The steps and the solutions of the problems in the method

Eventually, the "input" of the method are the algebraic descriptions of the type to be implemented and the import type, and some algorithms of the procedures. The "output," on the other hand, consists of theorems for each operation of the type. These are the assertions to be proven by one of the well-known methods.

Finally, let us add a note to all this. Following the "philosophy" of our method, we can declare that there are many other paths towards the code implementing the target type. An obvious path appears in Fig. 4: $SPEC_a^{(ax)} + SPEC_{imp}^{(ax)} \rightarrow SPEC(ax) \rightarrow SPEC_c^{(pp)}$. But, naturally, we can apply countless tiny steps, as well. It is very important, though, to check the correctness of each step. The concepts presented previously can be applied for the control of correctness.

Acknowledgements

The author would like to thank Prof. Emeritus László Varga and
Prof. László Kozma from University Eötvös Loránd for useful discussions on the material presented in this paper.

References

[1] Kozma,L.–Varga,L.: "*Adattípusok osztálya*", ELTE TTK, Informatikai Tanszékcsoport, Budapest, 2001

[2] Kozma,L.–Varga,L.: "*A szoftvertechnológia elméleti kérdései*", ELTE Eötvös Kiadó, Budapest, 2003

[3] Ehrig, H. and Mahr, B.: "*Fundamentals of Algebraic Specification 1 Equations and Initial Semantics*", Springer-Verlag Berlin Heidelberg, 1985

[4] Ehrig, H. and Mahr, B.: "*Fundamentals of Algebraic Specification 2 Module Specifications and Constraints*", Springer-Verlag Berlin Heidelberg, 1990

[5] Kurki-Suonio, R. and Mikkonen, Tommi: "*Liberating Object-Oriented Modelling from Program ming-Level Abstractions*", ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modelling Techniques 1997, pp. 115-121.

[6] Parisi-Presicce, F. and Pierantonio, A.: "*An Algebraic Theory of Class Specification*", ACM Transaction on Software Engineering and Methodology, Vol. 3, No. 2, 1994, pp. 166-199.

Postal address

Péter Szlávi
Informatics Methodology Group
University Eötvös Loránd
1/C., Pázmány P. sétány
H-1117 Budapest
Hungary