ABOUT METAMATHEMATICS OF COMPUTER PROGRAMMING

Valerie NOVITZKÁ
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, E-mail: Valerie.Novitzka@tuke.sk

SUMMARY

How to cancel the software crisis? To prove that programs are reliable. But it is possible only in mathematics. So, the programming has to use mathematics. In this paper we present basic mathematical concepts that enable to start discussion about the theory of programming in the framework of the classical mathematical logic, an axiomatic set theory, category theory, and universal algebra.

Keywords: *specification, program, category, institution*

1. INTRODUCTION

What is a program? The popular answer had been given by Niklaus Wirth as a title of his famous book *Data structures + Algorithms = Programs* [10]. But what means precisely the words „algorithms“ and „data structures“ ? We are not able to formulate the universal meaning of these concepts in a unique manner. However, in Wirth's book it is clear, that an algorithm means some sequence of statements written in the programming language Pascal, while data structures are definitions of types and declarations of variables of uniquely defined types written in the same programming language. But to prove correctness of an executed program, we need unambiguous formulation of the essential notions of data structures and algorithms. We take over from this book only the concept that description of algorithms and data structures are texts written in some artificial languages with mathematically defined meaning, i.e. in an unambiguous syntax and semantics. But we add to Wirth's concept that every step of algorithm including compiler, runtime environment and operating system has to be mathematically proved.

To characterize forming of a program we have

- to formulate a reasonable question for which we want to find a proved answer, and
- to describe with an unambiguous (not natural) language the process of obtaining a proved answer of the question in computer memory.

Firstly, we have to formulate all preconditions, which make the question answerable (*requirements specification)*. Secondly, we have to outline a *mapping*, which in a proved manner, truly work out an answer. We suppose, that the compiler, the runtime environment and the operating system are already proved to be correct, and we deal only with the correct mapping from the requirements specification written in a specification language, to the program text written in some programming language. (Providing the runtime environment can be able to check the correctness of execution, i.e. the dynamic semantics and invariants, the stacks, the queues, the trees and other dynamic data structures.) We emphasize that this mapping is realized by a human programmer (maybe with the assistence of a computer in the framework of a correct operating system). Of course, we suppose that the specified question is decidable (in the metamathematical sense) and the compiled program, the runtime environment and the operating system (in the computer) together have a reasonable complexity. Without these conditions the concept of executed program in the computer memory has no unique meaning.

We still note that we do not bind our considerations to any concrete syntax and semantics of specification and/or programming languages. We only suppose that the syntax is unambiguous and the semantics is mathematically formulated. Some authors called the concrete forms of our mapping as transformation [3,4]. But our mapping is a generalized transformation, by its mathematical foundation we want to avoid the pittfalls of the software crisis.

In this paper we talk about programming in a mathematical language, because we want to prove that an executed and terminated program in a computer memory truly answers a reasonable human question.

2. CATEGORIES

Every programmer knows that the design of a large scale program is constructed by frequent application of mappings. Because we would like to formulate mathematically the reasoning about programming process, we need mathematical notions by which this mapping is constructed. Such a notion is category. We begin with description of *object*. When we phenomenologically describe our world, we can emphasize some events or things of it and we decide about them that they have some kind of „individuality“, i.e. they differ from others in some sense. We call such an event or a thing as object. This description of objects includes the fact that they may not exist yet, but it is possible to

create them. Therefore the phrase: „an object exists“ actually means that it is possible to create it [9]. Such notion of object is useful for us because during the programming process we really create new objects from ready ones.

We suppose that there are some already created objects which have a common property, and there are no promoted objects between them. Such objects form a *collection.* Let we have a collection where every ob-ject has an uniquely defined property and every one can be promoted from other objects. Such collection we call a *set*. The notion of set can be axiomated. Elements of a set X are objects satisfying the common property “object is an element of the object“, written $x \in X$, which promotes a set object X from an element objects denoted by x. A *class* is a collection of objects with loosely defined common property. A class is not necessary a set.

In the programming process there are frequently such situations when programmers are interested only in special kinds of mappings between classes (or sets), called morphisms, and they are not interested in actual structure of the domains and codomains of them. Therefore we introduce the known concept of category [1,8] that we use in the defining rather sophisticated concepts needed for exact description of the program development process.

Definition 1: A category $\boldsymbol{C}$ is a quadruple $\boldsymbol{C=(C_{Obj}, hom_C, id_C, ;)}$, where

- $\boldsymbol{C_{Obj}}$ is a class of *objects*;
- $\boldsymbol{hom_C}$ is a set of *category morphisms* $f:A\to B$, for all objects $A,B \in \boldsymbol{C_{Obj}}$; for any objects A,B, we denote by $hom_C(A,B)$ the set of all morphisms between them;
- $\boldsymbol{id_C}$ is a set of *identity morphisms* $id_A:A\to A$, for every object $A \in \boldsymbol{C_{Obj}}$;
- $\boldsymbol{;}$ is an operator called *morphism composition* which assigns to two morphisms $f:A\to B$ and $g:B\to C$, where $A,B \in \boldsymbol{C_{Obj}}$ a composite morphism $g\,;f:A\to C$ such that $g\,;f \in hom_C(A,C)$.

□

The components of a category $\boldsymbol{C}$ are subjects to the following properties:

1. for each morphism $f:A\to B$, $A.B \in \boldsymbol{C_{Obj}}$ it holds $id_B\,;f=f=f\,;id_A$;
2. the composition of category morphisms is associative;
3. the sets $hom_C(A,B)$, for any objects $A,B \in \boldsymbol{C_{Obj}}$ are pairwise disjoint.

Now we introduce several examples of categories that are useful for our purposes.

Example 1: *Category* ***Set*** *of sets.*
The category of sets is $\boldsymbol{Set=(Set_{Obj}, hom_{Set}, id_{Set}, ;)}$, where

- $\boldsymbol{Set_{Obj}}$ is the class of all sets;
- $\boldsymbol{hom_{Set}}$ is the set of of mapping sets from A to B, i.e. $\boldsymbol{hom_{Set}}(A,B)=\{f:A\to B \mid A.B \in Set_{Obj}\}$ for every two sets A,B;
- $\boldsymbol{id_{Set}}$ is the set of identity mappings $id_A:A\to A$, for every set A;
- $\boldsymbol{;}$ is the operator of mapping composition between sets.

It is clear that components of ***Set*** satisfy category properties, i.e. we can say that ***Set*** is a category.

□

Let $\boldsymbol{C}$ be a category. Special cases of $\boldsymbol{C}$ are the following:

- if $\boldsymbol{C}$ contains exactly one object, it is essentially a monoid;
- if the sets $hom_C(A,B)$ of morphisms between any two objects A,B have at most one element, then $\boldsymbol{C}$ is essentially a preordered class;
- if $\boldsymbol{C}$ consists only of objects without morphisms, i.e. for any objects $A,B \in \boldsymbol{C_{Obj}}$, $hom_C(A,B)=\varnothing$, $\boldsymbol{C}$ is called *discrete category*. Clearly, a discrete category is a class of objects.
- if $\boldsymbol{C_{Obj}}$ is empty class, then such category is empty.

Example 2: *Sequential automaton in the category* ***Set***.

We can nested in the category ***Set*** the concept of sequential automaton.

A sequential S-automaton [2] $P=(Q, \delta, G, \gamma, q_0)$ is a device that is at one state $q \in Q$ and, receiving an input signal ς from the input alphabet S, it changes q to another state q' and simultaneously emits an output signal from the output alphabet G. Formally,

- Q is the set of states; G is the output alphabet;
- $\delta: Q\times S\to Q$ is the next-state mapping, i.e. each state q and input signal ς determine the next state $q'=\delta(q,\varsigma)$;
- $\gamma: Q\to G$ is an output mapping;
- q_0 is a special element of Q, the initial state of P.

Let $P=(Q, \delta, G, \gamma, q_0)$ and $P'=(Q',\delta',G',\gamma',q_0')$ be S-automata. A morphism from P to P' is a pair of mappings $(f, f_{out}): P\to P'$, where

$f:Q\to Q'$ and $f_{out}:G\to G'$

such that $f_{out}\,;\gamma=\gamma'$ and f preserves the initial state,

$f(q_0)=q_0'$.

Because Q, S and G are sets, δ and γ are mappings between sets, a sequential S-automaton can be depicted in the category *Set* as it is illustrated by the diagram in Fig.1, where the set $I=\{0\}$ is a singleton such that the initial state $q_0=\lambda(0) \in Q$.

□

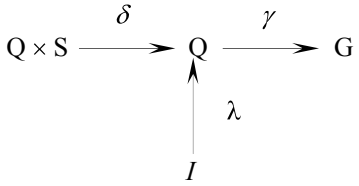

Fig. 1 *S-automaton in Set*

Very useful concept in category theory is the *principle of duality*. Let $\boldsymbol{C}$ be a category. We get a dual (or opposite) category $\boldsymbol{C}^{opp}$ by changing the direction of all category morphisms in $\boldsymbol{C}$. Every concept and every theorem in category theory comes with its dual version, where all morphisms are reversed.
Morphisms between categories are called functors.

Definition 2: Let $\boldsymbol{C}$ and $\boldsymbol{C}'$ be categories. A *functor* $F:\boldsymbol{C}\rightarrow\boldsymbol{C}'$ consists of

- a mapping $F: \boldsymbol{C}_{Obj}\rightarrow\boldsymbol{C}'_{Obj}$; and
- a set of mappings $F:hom_C(A,B)\rightarrow hom_{C'}(F(A), F(B))$, where A and B range over $\boldsymbol{C}_{Obj}$, so that for every A holds $F(id_A)=id_{F(A)}$ and $F(g);F(f)=F(g;f)$ for any morphisms $f:A\rightarrow B$ and $g:B\rightarrow C$ in $\boldsymbol{C}$.

□

We note here that in our theory of programming a functor may create objects and morphisms of the target category from the ones of the original category in a mathematically proved manner.

An *identity functor Id* on an arbitrary category $\boldsymbol{C}$ consists of an identity morphism on the class $\boldsymbol{C}_{Obj}$ and of the set of identity mappings on $\boldsymbol{hom}_C$.

Functors are defined as morphisms between categories. It is trivial to prove that they are closed under composition, which is associative, because it is the composition of functions between classes. We have introduced identity functor, too. Therefore we can consider about category of categories. But we forbit the existence of the category of all categories (Russell's paradox). To avoid this situation we consider the *category of small categories*, where small category is such, which objects form a set. Then the category $\boldsymbol{Cat}$ of small categories consists of the class $\boldsymbol{Cat}_{Obj}$ of small categories as objects and of the set $\boldsymbol{hom}_{Cat}$ of functors between them as category morphisms. For every object $\boldsymbol{C}$ from the category $\boldsymbol{Cat}$, the identity functor Id_C is the identity morphism and $\boldsymbol{id}_{Cat}$ is the set of them. Composition of category morphisms is the composition of functors, which is associative. We can say that $\boldsymbol{Cat}$ is a category, but it is *not* a small category.

Let $\boldsymbol{C}$ be an arbitrary category. A special case of functor is the functor $F_C:\boldsymbol{C}\rightarrow\boldsymbol{Set}$, which assigns to every object $A\in\boldsymbol{C}_{Obj}$ its underlying set (without structure) $X\in\boldsymbol{Set}_{Obj}$, such that

$$F_C(A)=X$$

and to every category morphism $f:A\rightarrow B$, $A,B\in\boldsymbol{C}_{Obj}$ a mapping $p:X\rightarrow Y$ defined by

$$F_C(f)=p,$$

where $X=F_C(A)$ and $Y=F_C(B)$.
Functor F_C maps a structured category's objects to their underlying sets, i.e. it „forgets" their structures. Therefore F_C is called *forgetful functor*.

Finally, we introduce the notion of object-free category, which will be very interesting in the theory of programming, because it deals only with morphisms.
An *object-free* category is a partial binary algebra

$$\boldsymbol{M} = (M, ;)$$

where members of M are morphisms satisfying the following conditions:

- composition of morphisms is associative;
- for every morphism f there exist units u_1, u_2 such that $u_1 ; f$ and $f ; u_2$ are defined;
- for any pair of units (u_1 , u_2) the class $hom(u_1 , u_2)$ is a set.

For every category $\boldsymbol{C}$ we can construct corresponding object-free category

$$\boldsymbol{M}_{\boldsymbol{C}} = (\boldsymbol{hom}_{\boldsymbol{C}} , ;).$$

Functors can also be considered as objects. There-fore it is possible to consider morphisms between func-tors called natural transformations.

Definition 3: Let $F,G: \boldsymbol{C}\rightarrow\boldsymbol{C}'$ be functors. A *natural transformation* $\tau: F\rightarrow G$ from F to G is

- a class of morphisms $(\tau_A: F(A)\rightarrow G(A))_{A\in C}$ between images of every object from $\boldsymbol{C}$ under functors F and G, and
- for every morphism $f:A\rightarrow B$ in $\boldsymbol{C}$ it holds

  $$\tau_A ; G(f) = F(f) ; \tau_B$$

  i.e. the diagram in Fig.2 commutes.

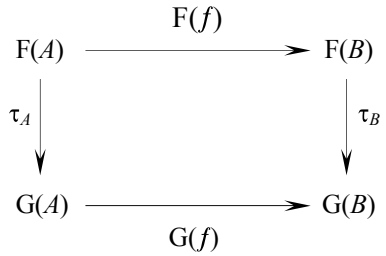

Fig. 2 *Natural transformation*

□

3. INSTITUTIONS

In this section we introduce useful logical and mathematical notion that enables us to deal with computer programming in the mathematically tractable manner.

One of the most significant results of Software Engineering is the division of a large scale program into intelectually managable modules called *data abstractions*. A data abstraction contains parameters and local data needed for related procedures and functions and enables to import and/or export them from or to another data abstractions. We emphasize here that the procedures and functions inside a data abstraction, i.e. local ones of it, must have unique properties to be these procedures and functions executable in the computer memory.

We recall the algebraic definition of a scheme of data abstraction that we can formulate in a conservative extension of an axiomatic set theory.

Definition 4: A *signature* Σ is a triad $\Sigma=(S,O,P)$ consisting of

- S, a linear ordered finite set of *sorts*;
- O, a finite set of (total or partial) function names of the form $f:<s_1,...,s_n>\to s$, where n is the arity of f, $<s_1,...s_n>\to s$ is its profile and $s_1,...s_n,s\in S$;
- P, a finite set of predicate names of the form $p:<s_1,...s_n>$, where n is the arity of p, $s_1,...s_n\in S$.

□

Let Σ and Σ' be signatures. *Signature morphism* $\sigma: \Sigma\to\Sigma'$ maps sorts, function names and predicate names from Σ to the corresponding ones from Σ', so that it preserves linear ordering of sorts, function profiles and predicate arities.

Example 3: A simple signature for natural numbers can be of the form

$$\Sigma_{nat}=(S_{nat},O_{nat},P_{nat})$$

where

$S_{nat}=\{nat\}$

$O_{nat}=\{zero:\to nat,\ succ: nat\to nat\}$

$P_{nat}=\{\_\le\_: <nat, nat>\}$

□

Now we briefly introduce the classical first order predicate logic, which enables us to construct closed formulae. Symbols of this logic are:

- *variables* of different sorts grouped into disjunct classes;
- *predicate names* with their arities;
- *function names* with their profiles. Function names with zero arities are *constants* of some sorts;
- *logical connectives* $\Rightarrow, \neg, \&, \vee, \Leftrightarrow$;
- *quantifiers* $\forall$ and $\exists$;
- *auxiliary symbols*, e.g. *(* and *)*.

Terms are formed by repeated application of the following two rules:

- every variable and constant is a term of some sort;
- if $f:<s_1,...s_n>\to s$ is a function name and $t_1,...t_n$ are terms of sorts $s_1,...s_n$, respectively, then $f(t_1,...t_n)$ is also a term of the sort s.

Formulae are created by repeated application of the following three rules:

- if $t_1,...t_n$ are terms of sorts $s_1,...s_n$, respectively, and $p:<s_1,...s_n>$ is a predicate name, then $p(t_1,...t_n)$ is a *basic* formula;
- if ψ_1 and ψ_2 are formulae, then also $\psi_1\Rightarrow\psi_2$, $\neg\psi_1$, $\psi_1\&\psi_2$, $\psi_1\vee\psi_2$ and $\psi_1\Leftrightarrow\psi_2$ are formulae;
- if ψ is a formula and $x:s$ is a variable of a sort s, then also $(\forall x)\psi$ and $(\exists x)\psi$ are formulae.

Every variable in a basic formula is free. Logical connectives do not change the freeness of variables. Quantifiers bind their variables, i.e. a variable x is bound in the formulae $(\forall x)\psi$ and $(\exists x)\psi$. A formula ψ in which all its variables are bound is *closed formula*. Closed formulae can be evaluated to be true or false, i.e. they are sentences. For instance, the following closed formula is sentence (more precisely, the power axiom of the Zermelo-Fraenkel axiomatic set theory):

$$(\forall x)(\exists y)(\forall z)(z\in y\Leftrightarrow(\forall u)(u\in z\Rightarrow u\in x))$$

because the variables x,y,z,u are bound.

Let Σ be a signature. We suppose that its sorts, function names and predicate names correspond with some symbols of our logic. So, we can formulate closed formulae evaluated to true sentences, which consist of the predicate names from Σ. They characterize some well-formed properties of function names from the signature. Such closed formulae we call Σ-sentences. The set of true Σ-sentences we denote by $\mathbf{\Phi_\Sigma}$. So, we can express a *specification* of a data abstraction as a pair

$$Spec=(\Sigma, \mathbf{\Phi_\Sigma}).$$

We construct the class $\boldsymbol{Sign_{Obj}}$ of signatures. We denote by $\boldsymbol{hom_{Sign}}$ the set of all signature morphisms between elements of this class, and by $\boldsymbol{id_{Sign}}$ the set of all identical signature morphisms $id_\Sigma: \Sigma\to\Sigma$, for every $\Sigma\in \boldsymbol{Sign_{Obj}}$. Because the composition of signature morphisms is closed in $\boldsymbol{Sign_{Obj}}$ and associative,

$$\boldsymbol{Sign=(Sign_{Obj}, hom_{Sign}, id_{Sign}, ;)}$$

is the category of signatures.

For any signature Σ from the category $\boldsymbol{Sign}$ we can construct a Σ-algebra as follows.

Definition 5: Let $\Sigma=(S, O, P)$ be a signature. A Σ-algebra is

$$A=(S_A,O_A,P_A),$$

where

- S_A is a class of data sets, such that the sorts from S are bijectively mapped to the sets from S_A.
- O_A is the set of (total or partial) functions (algebraic operations) named by (total or partial) function names from O. The domains and codomains of the functions come from the profiles of the corresponding function names from O.
- $P_A = \mathbf{\Phi_\Sigma}$ as defined above. We call these true Σ-sentences also *Σ-axioms*.

□

Example 4: One of the possible Σ_{nat}-algebra for the signature Σ_{nat} introduced in the Example 3 can be

$$A_N=(S_N, O_N, P_N),$$

where

- $S_N= N$ is the set of natural numbers;
- O_N contains the functions $zero_N=0$ and $succ_N(n)=n+1$, for a variable n ranging over the set N;
- P_N contains the relation defined by the true Σ_{nat}-axiom

$$(\forall n_1)(\exists n_2)(n_2 = succ_N(n_1))$$

□

From Σ-axioms we can derive other true closed Σ-formulae by the following way. Let $\mathbf{\Phi}_\Sigma$ be a set of Σ-axioms from a Σ-algebra A. A Σ-formula φ is derivable from $\mathbf{\Phi}_\Sigma$ in A if there exists a sequence of true closed Σ-formulae

$$\psi_1, \psi_2, \ldots, \psi_k$$

such that

- ψ_k is φ, and
- every ψ_i, $i<k$, is either a Σ-axiom or it can be derived from the previous Σ-formulae by the application of the following two rules:
  i) if ψ_1 and $\psi_1 \Rightarrow \psi_2$ are true closed formulae, then also ψ_2 is true closed formula (modus ponens);
  ii) if ψ is a true closed formula and x is any variable, then also $(\forall x)\psi$ is true closed formula (generalization rule).

Such sequence we call a *derivation* for φ in A. If there exists a derivation for a Σ-formula φ in A, we say that φ is *satisfied* in the Σ-algebra A, denoted by $A \models_\Sigma \varphi$. Such Σ-algebra in which we still have satisfied closed Σ-formulae we call Σ-*model*.

We define the functor *Sen:* $\mathbf{Sign \rightarrow Set}$ from the category of signatures to the category of sets, which assigns to every signature Σ a set of true Σ-sentences and to every signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ the mapping *trans-lation* of Σ-sentences that replaces all symbols in a Σ-sentence ψ with their images from Σ' under σ.

The class of the Σ-models together with the set of homomorphisms between them form the category $\mathbf{Mod(\Sigma)}$ of Σ-models.

Let $\sigma: \Sigma \rightarrow \Sigma'$ be a signature morphism and A' a Σ'-model. A *reduct* of A' with respect to (w.r.t) σ is the Σ-model

$$A'|_\sigma = (S'_{A|\sigma}, O'_{A|\sigma}, P'_{A|\sigma})$$

where

- $S'_{A|\sigma}$ is the class of data sets whose corresponding sorts are counter images of the sorts from S' w.r.t. σ,
- $O'_{A|\sigma}$ is the set of functions whose corresponding function names are counter images of the function names from O' w.r.t. σ,
- $P'_{A|\sigma}$ is the set of Σ'-sentences containing predicates named by counter images of the predicate names from P' w.r.t. σ.

Reduct functor $\_|_\sigma$: $\mathbf{Mod(\Sigma')} \rightarrow \mathbf{Mod(\Sigma)}$ from the category of Σ'-models to the category of Σ-models w.r.t. the signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ maps

- each Σ'-model A' to its reduct, the Σ-model $A'|_\sigma$, and
- each Σ'-homomorphism between Σ'-models to Σ-homomorphism between the corresponding reducts.

We can define the functor *Mod:* $\mathbf{Sign^{opp} \rightarrow Cat}$ from the dual category of signatures to the category of small categories that assigns to every object Σ the category $\mathbf{Mod(\Sigma)}$ of Σ-models and to every signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ the reduct functor $\_|_\sigma$: $\mathbf{Mod(\Sigma') \rightarrow Mod(\Sigma)}$.

Now we have defined all neccessary components of institution. An *institution* I is a quadruple

$$\mathbf{I} = (\mathbf{Sign}, Sen, Mod, \models)$$

where

- $\mathbf{Sign}$ is the category of signatures, *Sen* and *Mod* are functors as defined above,
- $\models$ is a set of satisfaction relations $\models_\Sigma$ for every signature Σ from $\mathbf{Sign}$,
- if $\sigma: \Sigma \rightarrow \Sigma'$ is a signature morphism from $\mathbf{Sign}$ and ψ is a Σ-sentence, then it holds the following equi-valence:

$$A' \models_{\Sigma'} Sen(\sigma)(\psi) \Leftrightarrow Mod(\sigma)(A') \models_\Sigma \psi$$

i.e. ψ is satisfied in the reduct $A'|_\sigma$ iff its translation w.r.t. σ is satisfied in A.

We have shown that institutions meaningfuly formalize the requirements specifications. But during a design and execution of a program in computer memory, we necessary construct the following steps, too: a program, a machine code before execution, and state of the computer memory after terminating of a program. Can we formalize also a program written in some programming language, a program as a sequence of machine instructions, and state of the memory after terminating machine instructions sequence? Yes, the classical mathematical logic, pure mathematics based on an axiomatic set theory, Theoretical Computer Science and Software Engineering are able to formalize and prove statements about these stages of programming process.

Our example of sequential automata confirms that. But the programmer from the underlying requirements specification really constructs a program; a compiler, librarians, linkage editors and loaders really construct the sequence of executable machine instructions; and finally, a runtime environment and operating system really execute this sequence and stop it provided this program is terminated. These „constructions" are mappings of institutions that also a programmer, who is well-educated mathematician, can work out and he can formalize true proved statements about the correctness of the whole programming process. For brevity, these mappings we will call *arrows* and we suppose that they are mathematically tractable entities. As a simple example (based on the similar one in [5]) of such an arrow follows.

Example 5: Let $\mathbf{I} = (\mathbf{Sign}, Sen, Mod, \models)$ be an institution. We construct

- a functor $\mathcal{G}$: $\mathbf{Sign \rightarrow Sign'}$ creating a new category $\mathbf{Sign'}$ of signatures so that we construct

to every signature Σ from ***Sign*** a signature Σ' and to every signature morphism $\sigma: \Sigma_1 \to \Sigma_2$ from ***Sign*** a morphism $\sigma': \vartheta(\Sigma_1) \to \vartheta(\Sigma_2)$. Because $\vartheta(\Sigma_1)$ and $\vartheta(\Sigma_2)$ are signatures, σ' is a signature morphism. It is trivial to prove that so constructed ***Sign'*** is the category with objects $\vartheta(\Sigma)$ and category morphisms σ' between them.

- a natural transformation $\mu^{Mod}: Mod \to Mod'; \vartheta$, i.e. a set of morphisms

$$\mu_{\Sigma}^{Mod}: \mathbf{Mod(\Sigma)} \to \mathbf{Mod'}(\vartheta(\Sigma))$$

for every signature Σ from ***Sign***, such that it constructs for every Σ-model A a $\vartheta(\Sigma)$-model A', and for every reduct functor $\_|_{\sigma}$ it constructs a reduct functor $\_|_{\vartheta(\sigma)}$.

We can formulate Σ'-sentences φ' for every signature Σ' from ***Sign'*** satisfied (i.e. proved) in Σ'-models A', $A' \models_{\Sigma'} \varphi$, so that there exists

- a natural transformation $\mu^{Sen}: Sen'; \vartheta \to Sen$ i.e. a set of morphisms

$$\mu_{\Sigma'}^{Sen}: Sen'(\vartheta(\Sigma)) \to Sen(\Sigma)$$

for every signature Σ from ***Sign***; and for every Σ-model.
A from ***Mod(Σ)*** the following equivalence holds:

$$A \models_{\Sigma} \mu_{\Sigma'}^{Sen}(\varphi') \Leftrightarrow \mu_{\Sigma}^{Mod}(A) \models'_{\vartheta(\Sigma)} \varphi'$$

The construction described above ensures that

$$\mathbf{I'} = (\mathbf{Sign'}, Sen', Mod', \models')$$

is an institution and we call the morphism

$$\mu = (\vartheta, \mu^{Mod}, \mu^{Sen}): \mathbf{I} \to \mathbf{I'}$$

institution morphism.

□

4. CONCLUSION

Finally, we should like to emphasize that we are returning to the Polya's idea of problem solving in mathematics [6,7]. From the mathematics's point of view the programming is such a problem solving. It is clear in the case how the programmers work out their programs respecting in every step their requirements specifications. But it has been clear also when the programmers create special but correct programs for compilation, etc. We are sure that the software crisis should be stepped over and the mathematically well-educated programmers have to construct correct programs.

REFERENCES

[1] J.Adámek, H.Herrlich, G.E.Strecker: *Abstract and Concrete Categories,* Wiley& Sons, New York, 1989

[2] J.Adámek, V.Trnková: *Automata and Algebras in Categories*, Kluwer Academic Publishers Group, Dordrecht, 1990

[3] B.Krieg-Brückner: Programmentwicklung durch Spezifikation und Transformation, Research Project KORSO, Univ.Bremen, 1994

[4] M.Hoffmann, B.Krieg-Brückner: *Program Development* by Specification and *Transformation: Methodology-Language Family -System*, Springer, LNCS 680, 1993

[5] J.A.Goguen, R.M.Burstall: Introducing institutions, *Proc. Logics of Programming Workshop*, Springer, LNCS 164, 1984, pp.221-256

[6] G.Polya: *How to solve it?*, Princeton Univ. Press, 1946

[7] G.Polya: *Mathematics and Plausible reasoning,* Princeton Univ.Press, 1954

[8] L.Schröder: Categories: a free tour, In: A.Melton, J.Koslowski, eds: *Categorical Perspectives*, Birkhäuser, Basel, 2001, pp.1-27

[9] P.Vopěnka: *Mathematics in the Alternative Set Theory,* Teubner Leipzig, 1979

[10] N.Wirth: *Data structures + Algorithms = Programs,* Prentice-Hall, Englewood Cliffs, 1975

BIOGRAPHY

Valerie Novitzká graduated (MSc.) at the Faculty of Sciences, the University of P.J.Šafárik, Košice in 1976. She defended her PhD. work "On Formal Semantics of ANNA" at the Hungarian Academy of Sciences, Budapest, in 1989. Now she work as assistant professor at the Department of Computers and Informatics at the Technical University in Košice. Her scientific research is focusing on theoretical computer science, especially the theory of programming and its metamathematics, and the semantics of programming and specification languages.