Annales Univ. Sci. Budapest., Sect. Comp. 23 (2004) 193-212

METAMATHEMATICAL FUNDAMENTAL CONCEPTS OF COMPUTER PROGRAMMING

V. Novitzká and B. Novitzky (Košice, Slovakia)

Abstract. In our paper we state that every reasonable question which has to be answered by a computer must have its mathematical theory as a basis. This basis is needed for proving the algorithm answering the question. The algorithm and data can be specified and programmed by languages. The metamathematics of these languages and programming can be formulated by concepts of the mathematical logic, set theory, category theory and universal algebra.

1. Introduction

What is a program? The most popular answer had been given by Niklaus Wirth as the title of his famous book *Data structures + Algorithms = Programs* [14]. But what mean precisely the words 'algorithms' and 'data structures'? We are not able to formulate the general meaning of these concepts in an explicit 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, from the metamathematical point of view, 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.

To characterize exactly the forming of a program we have

• to formulate a reasonable (scientific) question for which we want to find a reasonable (logically) proved answer, and

• to describe the algorithm and data structures for finding the proved answer for the specified question.

The languages used for the first task we call *specification languages* as e.g. CASL [4] and Larch [8] and the languages for the second task are (procedural, logical or functional) *programming languages*, as e.g. Ada [7], Prolog [3] and Haskell [9]. The task of the program forming, that we call *programming*, is to transit from the *specification* of the question in a mathematically precise manner to the *program* which together with data structures algorithmically answer the specified question. The forming such a program has to guarantee that the specified problem is decidable (solvable), that the methods of problem solving have reasonable complexity and that the results of problem solving are mathematically proved. Without these conditions the universal concept of programs has no mathematically precise meaning.

We have to consider that a specification (or a program) can be heterogeneous, i.e. it may be written in many specification (or programming) languages and transitions from specifications to programs have to reflect this heterogeneity. Therefore the programming is actually a transition from specifications to programs which in the PROSPECTRA project [5] is called transformation. We use for transitions more mathematical notions *arrows*.

So, in the metamathematics of programming we have to characterize the (original) theory of specification, the (target) theory of final program and an arrow from the original to the target one. In this paper we first give a foundation of the metamathematical notion of programming and then we shortly introduce an example of a concrete theory of programming using this metamathematics.

2. Categories in programming

In the next few definitions we use some ideas of the alternative set theory of Petr Vopěnka [13]. We begin with description of the notion '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 an object such an event or a thing. 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. It is the fact, in which our notion of object differs from the approach of elements of sets in Cantor's set theory. Such notion of object is useful in the metamathematics of programming 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 us have a collection where every object has a uniquely defined property and everyone can be promoted from other objects. Such collection we call a *set*. For the so defined sets we can apply Cantor's set theory or some its axiomatized version. Elements of a set X are objects satisfying the common property 'object is an element of the object', written as $x \in X$ which promotes a set object X from an element object denoted by x. An unordered pair is a set $\{x, y\}$, where x and y are objects. An ordered pair $\langle x, y\rangle$ can be defined as the set of two sets by

$$\langle x, y\rangle = \{\{x\}, \{x, y\}\}$$

which uniquely determines objects x and y and the property: 'x is the first and y is the second element of the ordered pair $\langle x, y\rangle$'. This definition can be extended to ordered n-tuples $\langle x_1, \ldots, x_n\rangle$.

A *class* is a collection of objects with loosely defined common property. A class is not necessarily a set, however we define also for classes the property: 'object is an element of a class'. Every set is a class; a class which is not a set is called a *proper class*. Let X and Y be classes (or sets), we say that X is a *subclass* (or *subset*) of Y, $X \subseteq Y$, if every element of X is also an element of Y. A class X is a *semiset*, if there exists a set Y such that $X \subseteq Y$.

We assume that all elements of sets and classes were already created. We define the notion of a *semicollection* for which we can create and put in new objects. When we already do not intend to put new objects into the semicollection, then it becomes a collection. Further, if every object x has a loose property $\varphi(x)$, then such collection $\{x|\varphi(x)\}$ with already created objects becomes a class. In a similar way we can define from such a class also a set. Because in our metamathematics of programming we deal with objects created within the program development process, in some cases we use the notion *semicollection* which better reflects this property. The property 'object R is a binary relation' we define as follows: R is a class and for every $y \in R$ there are y_1, y_2 such that $y = \langle y_1, y_2\rangle$. We can define the usual properties of binary relations as reflexivity, symmetricity, antisymmetricity, transitivity and dichotomicity. Then a reflexive, symmetric and transitive relation is called equivalence; a reflexive, antisymmetric and transitive relation is called ordering; and an ordered and dichotomic relation is called linear ordering.

Let X, Y be classes. We define a property 'an object f is a *mapping* from X to Y', written $f : X \to Y$, as a binary relation which satisfies

$$(\forall x_1 \in X)(\forall y_1, y_2 \in Y)((\langle x_1, y_1\rangle \in f \;\&\; \langle x_1, y_2\rangle \in f) \Rightarrow y_1 = y_2)).$$

The class X is called domain, Y codomain of the mapping f.

We can use the notions defined above in the introduction of the concept of category. In the programming process there are frequently such situations when programmers are interested only in special kinds of mappings called morphisms and they are not interested in the actual structure of the domains and codomains of them. Therefore we introduce the known concept of category [1,12] that we use in defining rather sophisticated concepts needed for exact description of the program development process.

Definition 1. A category $\mathbf{C}$ is a quadruple

$$\mathbf{C} = (\mathbf{C_{Obj}}, \mathbf{hom_C}, \mathbf{id_C}, \circ),$$

where

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

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

1. for each morphism $f : A \rightarrow B,\ f \in hom_C(A, B),\ A, B \in \mathbf{C_{Obj}}$

$$id_B \circ f = f = f \circ id_A;$$

2. the composition of category morphisms is associative, i.e. for morphisms $f : A \rightarrow B,\ g : B \rightarrow C$ and $h : C \rightarrow D$, where $A, B, C, D \in C_{Obj}$

$$h \circ (g \circ f) = (h \circ g) \circ f;$$

3. the sets $hom_C(A, B)$ for any objects $A, B \in \mathbf{C_{Obj}}$ are pairwise disjoint.

The previous definition of category uses notions *object, class* and *morphism*. These notions are objects and classes in the sense of all set theories. A morphism is an ordered two element set. We use the ideas of the alternative set theory when we develop a new program text from the existing ones. Under a specification (or program) text, we mean a text written in some specification (or programming) language with modular and hierarchical structures.

Now we introduce (using Cantor's set theory) several examples of categories that are useful for our purposes.

Example 1. *Category* **Set** *of sets.*

The category of sets is

$$\mathbf{Set} = (\mathbf{Set_{Obj}}, \mathbf{hom_{Set}}, \mathbf{id_{Set}}, \circ),$$

where

- $\mathbf{Set_{Obj}}$ is the class of all sets;
- $\mathbf{hom_{Set}}$ is the set of mapping sets from A to B, i.e.

$$hom_{Set}(A, B) = \{f : A \to B \mid A, B \in \mathbf{Set_{Obj}}\}$$

for every two sets A, B;

- $\mathbf{id_{Set}}$ is the set of identity mapping $id_A : A \to A$ for every set A;
- $\circ$ is the operator of mapping composition between sets.

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

Let **C** be a category. There are some special cases of **C**:

- if **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 **C** is essentially a preordered class;
- if **C** consists only of objects without morphisms, i.e. for any objects $A, B \in \mathbf{C_{obj}}$, $hom_C(A, B) = \emptyset$, then **C** is called *discrete category.* Clearly, a discrete category is a class of objects;
- if $\mathbf{C_{Obj}}$ is empty class, then **C** is empty category.

Example 2. *Category of* Ω*-algebras.*

Let $\Omega = (n_i)_{i \in I}$ be a set of natural numbers n_i, $i \in I$. An Ω-algebra A is a pair

$$A = (X, (w_i)_{i \in I})$$

consisting of a set X and a set of mappings

$$w_i : X^{n_i} \to X$$

called n_i-ary operations on X. An Ω-homomorphism $f : A \to A'$, where $A = (X, (w_i)_{i \in I})$ and $A' = (X', (w'_i)_{i \in I})$, is a mapping $f : X \to X'$, such that $f \circ w_i = w'_i \circ f^{n_i}$ for every $i \in I$.

The category $\mathbf{Alg}(\Omega)$ of Ω-algebras consists of

- the class of Ω-algebras as objects;

- a set of Ω-homomorphisms between any two objects;
- a set of identical mappings from any object to itself;
- composition is usual composition of homomorphisms.

Composition of Ω-homomorphisms is also Ω-homomorphism and it is associative, therefore $\mathbf{Alg}(\Omega)$ is a category.

Example 3. *Category* **Top** *of topological spaces.*

A topological space is a pair $(X, \mathbf{I})$, where X is a set of points and $\mathbf{I}$ is an interior operation satisfying the following properties: Let A, B be sets, $A, B \subseteq X$. Then

(i1) $\mathbf{I}(A \cap B) = \mathbf{I}A \cap \mathbf{I}B$;
(i2) $\mathbf{I}A \subseteq A$;
(i3) $\mathbf{II}A = \mathbf{I}A$;
(i4) $\mathbf{I}X = X$.

A set $\mathbf{I}A$ is called an interior of A; and A is an open set if $A = \mathbf{I}A$.

Let X, Y be topological spaces. A mapping $f : X \to Y$ is *continuous* if for every open set $B \subseteq Y$ also its counter image under f, $f^{-1}(B) \subseteq X$ is open set. It is simple to prove that the composition of continuous mappings is also a continuous mapping [1].

A continuous mapping $id_X : A \to A$ is the identity on X if $id_X(A) = A$.

Then the category $\mathbf{Top} = (\mathbf{Top_{Obj}}, \mathbf{hom_{Top}}, \mathbf{id_{Top}}, \circ)$ of topological spaces consists of

- the class $\mathbf{Top_{Obj}}$ of topological spaces as objects;
- the set $\mathbf{hom_{Top}}$ of continuous mappings between all objects as category morphisms;
- the class $\mathbf{id_{Top}}$ of identities on all objects;
- composition denoted by operator $\circ$ is the composition of continuous mappings.

We will use the category of topological spaces in defining such subtheories of an arbitrary set theory in which we use some kind of 'continuity'. For instance, if we define in our theory a concept of probability as a normed measure on subsets of a topological space.

Example 4. *Sequential automaton in the category of sets.*

We can nest in the category **Set** of sets the concept of sequential automaton. This example is important for the reason that we would like to work out our metatheory of programming so general as it is possible. That means,

it could be enable to formalize also the concepts of programs not only for computers of von Neumann architectures.

A sequential S-automaton [2] $P = (Q, \delta, G, \gamma, q_0)$ is a device that is at a state $q \in Q$ and, receiving in 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

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

such that

$$f_{out} \circ \gamma = \gamma' \circ f \quad \text{and} \quad f \text{ preserves the initial state, } f(q_0) = q_0'.$$

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 *Figure 1*, where the set $\{0\}$ serves as domain for initialization mapping λ such that the initial state is $q_0 = \lambda(0) \in Q$.

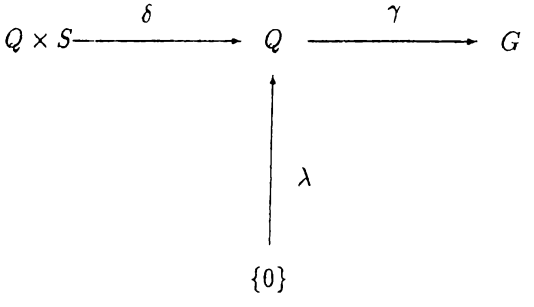

Figure 1. S-automaton in **Set**

A very useful concept in category theory is the *principle of duality.* Let $\mathbf{C}$ be a category. We get a dual (or opposite) category $\mathbf{C^{opp}}$ by changing the direction of all category morphisms in $\mathbf{C}$.

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 $\mathbf{C}$ and $\mathbf{C}'$ be categories. A *functor* $F : \mathbf{C} \to \mathbf{C}'$ consists of

- a mapping $F : \mathbf{C_{Obj}} \to \mathbf{C'_{Obj}}$;
- and a set of mappings $F : hom_C(A, B) \to hom_C(F(A), F(B))$, where A and B range over $\mathbf{C_{Obj}}$, so that for every $A \in \mathbf{C_{Obj}}$ it holds

$$F(id_A) = id_{F(A)}$$

and

$$F(g) \circ F(f) = F(g \circ f)$$

for any morphisms $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$.

We note here that in our metamathematics a functor may create objects and morphisms of the target category from the ones of the original category in a mathematically proved manner. This idea is useful in the cases when the objects of a category are some kind of specification or program written in some languages. In the next section we describe how to characterize program development in mathematically proved manner.

An *identity functor* Id on an arbitrary category $\mathbf{C}$ consists of identity morphism on the class $\mathbf{C_{Obj}}$ and of the set of identity mappings on $\mathbf{hom_C}$. Of course, in this case the target category $\mathbf{C}$ is already created.

In Example 4 above we have shown how a sequential automaton can be represented in the category of sets. By using the concept of functor, we can determine more general concept of so called F-automaton in an arbitrary category $\mathbf{C}$ as follows.

Example 5. *F-automaton.*

Let $\mathbf{C}$ be a category and let $F : \mathbf{C} \to \mathbf{C}$ be a functor on this category. We define F-automaton in the category $\mathbf{C}$ as a tuple

$$P = (Q, \delta, G, \gamma, I, \lambda),$$

where

• Q, G, I are objects of $\mathbf{C}$; Q is the state object, G is the output object and I is the initialization object;

• δ, γ, λ are morphisms in $\mathbf{C}$, $\delta : F(Q) \to Q$ is the next-state morphism, $\gamma : Q \to G$ is output morphism and $\lambda : I \to Q$ is initialization morphism.

If the functor F is defined on the category of sets, i.e. $F : \mathbf{Set} \to \mathbf{Set}$ by

$$F(X) = X \times S \quad \text{for every set} \quad X \in \mathbf{Set_{Obj}},$$
$$F(f) = f \circ id_S \quad \text{for every mapping} \quad f \text{ from } \mathbf{Set},$$

then such F-automaton is just sequential S-automaton.

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 category of categories. But we forbid the existence of the category of all categories (Russell's paradox). To avoid this situation we consider *category of small categories*, where small category is such a category whose objects form a set. Then the category **Cat** of small categories consists of the class $\mathbf{Cat_{Obj}}$ of small categories as objects and of the set $\mathbf{hom_{Cat}}$ of functors between them as category morphisms. For every object $\mathbf{C}$ from the category **Cat**, the identity functor Id_C is the identity morphism and $\mathbf{id_{Cat}}$ is the set of them. Composition of category morphisms is the composition of functors, which is associative. We can say that **Cat** is a category, but it is *not* a small category.

Let $\mathbf{C}$ be an arbitrary category. A special case of functor is the functor

$$F_C : \mathbf{C} \to \mathbf{Set}$$

from the category $\mathbf{C}$ to the category **Set** of sets, which assigns to every object $A \in \mathbf{C_{Obj}}$ its underlying set (without structure) $X \in \mathbf{Set_{Obj}}$, such that

$$F_C(A) = X$$

and to every category morphism $f : A \to B, A, B \in \mathbf{C_{Obj}}$ is a mapping $p : X \to Y$ defined by

$$F_C(f) = p, \quad \text{where} \quad X = F_C(A) \quad \text{and} \quad Y = F_C(B).$$

Functor F_C maps a structured category's objects to their underlying sets, i.e. it 'forgets' the structure of them. Therefore F_C is called *forgetful functor*. The forgetful functor actually creates its target.

Functors can also be considered as objects. Therefore it is possible to consider morphisms between functors called natural transformations.

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

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

$$\tau_A \circ G(f) = F(f) \circ \tau_B,$$

i.e. the diagram in *Figure 2* commutes.

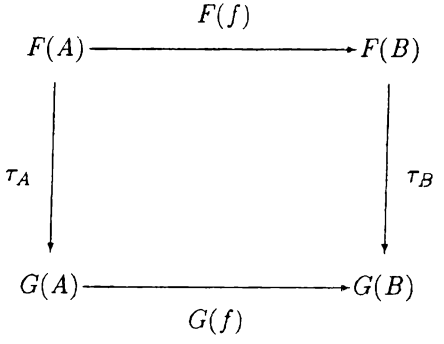

Figure 2. Natural transformation

3. Theories for languages

In this section we try to understand what is the programming, i.e. how we can specify a real question to be answered by computer and how to transit from this specification to the final program which actually answers the question. To do this transition we have to describe at least two mathematical theories: one for specifying a question and one for construction of a program which answers it, i.e. algorithmcially solves the problems included in it.

The practice of programming has proved that every algorithm and data forming a program has to be modularized to be intellectually managable, and that an optimal form of a modul is an abstract data type (ADT). It contains parameters and local data needed for related procedures and functions and enables to import and/or export them from and/or to another ADTs.

Many programming languages supporting ADTs were developed, e.g. Alphard [15] with a construction *form* and Ada with *generic packages*. But using an equivalent of ADTs is possible also in logical or functional programming languages. We have mentioned above that the specifications and programs can be heterogeneous. Of course, heterogeneous specifications and their programs necessarily form graphs, i.e. these specifications and programs are also hierarchical.

The syntax of an ADT we call *signature*, a known notion from universal algebra. We call semantics of an ADT as a set of *sentences*, characterizing the exact meaning of an ADT of the specified question or answering program. These sentences can be true of false, therefore we need some model in which we can test the satisfiability of them because we should like to bypass using the whole entailment. In the next text we define signatures, sentences and models from which we form the theories of specifications and programs.

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 $f : \langle s_1, \ldots, s_n \rangle \to s$, where n is the arity of f, $\langle s_1, \ldots, s_n \rangle \to s$ is its profile and $s_1, \ldots, s_n, s \in S$;
- P, a finite set of predicate names of the form $p : \langle s_1, \ldots, s_n \rangle$, where n is the arity of p, $s_1, \ldots, s_n \in S$.

A signature contains the names of components of an ADT. 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 6. A simple signature for natural numbers can be of the form

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

where

$$\begin{aligned} S_{nat} &= \{nat\}, \\ O_{nat} &= \{zero :\to nat, succ : nat \to nat\}, \\ P_{nat} &= \{\_ \leq \_ : \langle nat, nat \rangle\}. \end{aligned}$$

We construct the class $\mathbf{Sign_{Obj}}$ of signatures needed for specifying our question to be solved by a program. We denote by $\mathbf{hom_{Sign}}$ the set of all signature morphisms between elements of this class and by $\mathbf{id_{Sign}}$ the set of all identical signature morphisms $id_\Sigma : \Sigma \to \Sigma$ for every element $\Sigma \in \mathbf{Sign_{Obj}}$.

Because the composition of signature morphisms is closed in $\mathbf{Sign_{Obj}}$ and associative,

$$\mathbf{Sign} = (\mathbf{Sign_{Obj}}, \mathbf{hom_{Sign}}, \mathbf{id_{Sign}}, \circ)$$

is the category of signatures.

Now we formulate the formal language of a metamathematics of a set theory which serves for writing sentences. Symbols of this language are:

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

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

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

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

1. if $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, respectively, and $p : \langle s_1, \ldots, s_n \rangle$ is a predicate name, then $p(t_1, \ldots, t_n)$ is a *basic* formula;
2. if ψ_1 and ψ_2 are formulas, then also $\psi_1 \Rightarrow \psi_2$, $\neg\psi_1$, $\psi \wedge \psi_2$, $\psi_1 \vee \psi_2$ and $\psi_1 \Leftrightarrow \psi_2$ are formulas;
3. 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 formulas.

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 formulas $(\forall x)\psi$ and $(\exists x)\psi$. A formula ψ in which all its variables are bound is *complete formula*. Complete formulas can be evaluated as sentences, i.e. they can be true or false. For instance, the following complete 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 from the category **Sign**. The Σ-sentence is such complete formula that contains only symbols from Σ.

For any signature Σ as a syntax for an ADT we construct a Σ-model that is a model from universal algebra (called also many-sorted extended algebraic model) as follows.

Definition 5. Let $\Sigma = (S, O, P)$ be a signature. A Σ-model is an algebra $A = (S_A, O_A, P_A)$ where

- S_A is a class of data sets, such that the sorts from S are injectively mapped to the sets from S_A. The cardinality of S_A is at least such that the cardinality of S. Elements of sets from S_A can be denoted by variables.
- O_A is the set of (total or partial) functions (algebraic operations) named by (total or partial) function names from O. The domains and ranges of the functions come from the profiles of corresponding function names from O.
- P_A is the set of true Σ-sentences containing concrete predicates corresponding to the predicate names from P. These Σ-sentences express the algebraic properties of model functions. We call Σ-sentences also Σ-model axioms.

Example 7. One of the possible models for the signature Σ_{nat} introduced in the Example 6 can be the Σ_{nat}-algebra

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

where

$S_N =$**N** is the set of natural numbers;

O_N contains the functions $zero_N = 0$ and $succ_N = n + 1$, for a variable n ranging over the set **N**;

P_N contains the relation defined by the true sentence

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

From Σ-model axioms we can derive other true Σ-sentences by the following way. Let Ψ be a set of Σ-axioms from a Σ-model A. A Σ-sentence φ is true if

- there exists a sequence of the true Σ-sentences

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

- ψ_k is φ, and
- every ψ_i, $i < k$, is either a Σ-model axiom or it can be derived from the previous sentences by the application of the following two rules:

1. if ψ_1 and $\psi_1 \Rightarrow \psi_2$ are true sentences, then also ψ_2 is true sentence (modus ponens);

2. if ψ is a true sentence and x is any variable, then also $(\forall x)\psi$ is true sentence (generalization rule).

Such sequence $\psi_1, \psi_2, \ldots, \psi_k$ we call a *proof* for φ. If there exists a proof for a Σ-sentence φ, we say that φ is *satisfied in the* Σ*-model* A, denote by

$$A \models_\Sigma \varphi.$$

We define the functor Sen:**Sign**→**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 $\sigma : \Sigma \to \Sigma'$ the mapping *translation* of Σ-sentences, that replaces symbols in a Σ-sentences 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 $\sigma : \Sigma \to \Sigma'$ be a signature morphism and A' a Σ'-model. A *reduct* of A' with respect to σ is the Σ-model

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

where

- $S'_{A|\sigma}$ is the class of data sets whose corresponding sorts are counter images of the sorts from S' with respect to σ;
- $O'_{A|\sigma}$ is the set of functions whose corresponding function names are counter images of the function names from O' with respect to σ;
- $P'_{A|\sigma}$ is the set of Σ'-sentences containing predicates named by counter images of the predicate names from P' with respect to σ.

Reduct functor $\_|$: $\mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ from the category of Σ'-models to the category of Σ-models with respect to the signature morphism $\sigma : \Sigma \to \Sigma'$ 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.

Now we can define the functor Mod : $\mathbf{Sign}^{\mathbf{opp}} \to \mathbf{Cat}$ 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 $\sigma : \Sigma \to \Sigma'$ the reduct functor $\_|$: $\mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$.

Now we have defined all necessary components of institution [6]. An *institution* **I** is a quadruple

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

where

• **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 Σ from **Sign**,

• if $\sigma : \Sigma \to \Sigma'$ is a signature morphism from **Sign** and ψ is a Σ-sentence, then it holds the following equivalence

$$A' \models Sen(\sigma)(\psi) \Leftrightarrow Mod(\sigma)(A') \models \psi,$$

i.e. ψ is satisfied in the reduct $A'_{|\sigma}$ iff its translation with respect to σ is satisfied in A.

We now give some notes about the definition of institution. A signature Σ is the syntax of an ADT. The set of Σ-sentences contains the definition of semantics of the ADT and also definitions and theorems from that mathematical theory which is needed to formulate the facts for specifying question and/or proving the algorithm answering the question. In Σ-models we can prove new properties from Σ-sentences provided that between axioms of this model there are axioms of a suitable general axiomatic set theory.

As we know from the program construction, we are able to tell that the program development must have at least two institutions: (original) one for the specification of the question and (target) one for a final program fully answering the specified question. The main problem in formalization of program development is to define mathematically provable transitions of original institution for specification to the target institution for a final program. In this paper we are not concerned with concrete syntaxes of the specification and programming languages; defining them will be the subject of further research. We mention here only that there are several methods for constructing arrows by which it is possible to create the target institution from the original one.

Now we present several illustrative examples for arrows.

Example 8. Let **I**=(**Sign**,*Sen*, *Mod*, $\models$) be an institution. We construct

• a functor Θ : **Sign** $\to$ **Sign**$'$ creating a new category **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 $\sigma' : \Theta(\Sigma_1) \to \Theta(\Sigma_2)$. Because $\Theta(\Sigma_1)$ and $\Theta(\Sigma_2)$ are signatures, σ' is a signature

morphism. It is trivial to prove that such constructed $\mathbf{Sign}'$ is the category with object $\Theta(\Sigma)$ and category morphism σ' between them.

• A natural transformation $\mu^{Mod} : Mod \rightarrow Mod' \circ \Theta$, i.e. a set of morphisms

$$\mu^{Mod}_{\Sigma} : \mathbf{Mod}(\Sigma) \rightarrow \mathbf{Mod}'(\Theta(\Sigma))$$

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

Now we have a new syntax and new models of a new institution. We can formulate Σ'-sentences φ' for every signature Σ' from $\mathbf{Sign}'$ satisfied (i.e. provable) in Σ'-models $A', A' \models'_{\Sigma'} \varphi'$, so that there exists

• a natural transformation

$$\mu^{Sen} : Sen' \circ \Theta \rightarrow Sen,$$

i.e. a set of morphisms

$$\mu^{Sen}_{\Sigma'} : Sen'(\Theta(\Sigma)) \rightarrow Sen(\Sigma)$$

for every signature Σ from **Sign**; and for every Σ-*model* A from $\mathbf{Mod}(\Sigma)$ the following equivalence holds

$$A \models_{\Sigma} \ \mu^{Sen}_{\Sigma'}(\varphi') \Leftrightarrow \mu^{Mod}_{\Sigma}(A) \models'_{\Theta(\Sigma)} \varphi'.$$

The construction described above ensures that

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

is an institution and we call the morphism

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

as *institution morphism*.

Institution morphism enables construction of a new institution from an original one by enriching syntax and semantics of ADTs. This example we can simplify at two levels:

1. We do not consider morphisms between models, i.e. each $\mathbf{Mod}(\Sigma)$ is a discrete category, a class. In this case we can construct a target institution much easier, the satisfiability of sentences in models of the original isntitution is simply mapped to models in the target institution. Such simplified institution morphism is called *institution representation*.

2. We do not require also the holding of the satisfaction equivalence of institution morphisms. Then the constructed target institution is pure 'renaming' of constituents of the original one. Then we speak about *institution coding.*

Of course, institution representation and institution coding do not form enriched target institutions.

The traditional approach of algebraic specification of abstract data types uses the notion specification and/or program as a pair (Σ, Ψ), where signature Σ is its syntax and a set Ψ of Σ-sentences its semantics. Such syntax and semantics can be used for defining a language (for specifications and programs). In the following example we discuss the problem of arrows between institutions that arises from such considerations.

Example 9. Let $\mathbf{I}$=(**Sign**,$Sen, Mod, \models$) be an institution. We consider a pair $(\Sigma, \Psi)_I$, where Σ is a signature from **Sign**, syntax of an abstract data type, and Ψ is a set of Σ-sentences from $Sen(\Sigma)$, its semantics. We can define a category $\mathbf{ADT}(\mathbf{I})$ consisting of all such pairs $(\Sigma, \Psi)_I$ as objects and morphisms

$$\xi : (\Sigma_1, \Psi_1) \to (\Sigma_2, \Psi_2)$$

such that $\xi : \Sigma_1 \to \Sigma_2$ is a signature morphism and $Sen(\xi)(\Psi_1) \subseteq \Psi_2$.

To take into account this approach we use as the base for arrows building new institution the concept of mapping institution introduced by Messeguer in [10].

We construct

• a functor Θ:$\mathbf{ADT}(\mathbf{I})\to\mathbf{ADT}$ $(\mathbf{I}')$, that constructs a new pair (Σ', Ψ'), i.e. syntax and semantics of abstract data type in a new theory $\mathbf{I}'$ and maps every morphism ξ to the morphism between images of Σ and Ψ, respectively, under Θ;

• natural transformations

$$\nu^{Sen} : Sen \to Sen' \circ \Theta,$$
$$\nu^{Mod} : Mod' \circ \Theta \to Mod$$

such that for every Σ-sentence $\varphi \in Sen(\Sigma)$, model $A' \in \mathbf{Mod}'(\Theta(\Sigma, \emptyset))$ the following equivalence holds

$$A' \models'_{\Sigma'},\ \nu^{Sen}_{\Sigma}(\varphi) \Leftrightarrow \nu^{Mod}_{(\Sigma,\emptyset)}(A') \models_{\Sigma} \varphi.$$

Then $\nu = (\Theta, \nu^{Sen}, \nu^{Mod})$: $\mathbf{I}\to \mathbf{I}'$ is called *map of institutions.*

Let us answer a nontrivial question. Let us specify it by a large scale heterogenous and hierarchical specification in an algebraic specification language like specification language. It would be very impracticle to construct immediately one arrow from this original institution to a target one containing the category of all signatures (ADTs) and the set of all sentences describing the semantics of every signature, and all necessary definitions and/or theorems of each mathematical theory containing the basis for proving the mathematical correctness of the answer for the specified question. Of course, the mathematical theories mentioned in the semantics above are necessarily formulated as subtheories in a suitable axiomatic set theory. This fact implies that the models contained in this institution must have axioms partly from this set theory, partly also from the used subtheories to make easier the proofs of some new theorems which will be necessary in the proving of algorithmical steps in the construction of result program answering the specified question. We suppose, that data abstractions of this institution form a graph, i.e. they are also heterogeneous and hierarchical and their concrete texts will be written in some kind of modular language. So it is reasonable to define more than two institutions and more than one arrow between neighbouring institutions from which the first will be the original and the second the target.

So, we divided the answer of a complicated question into a sequence of steps in the sequence of specification and/or program theories in the sequence of the according institutions and also in a sequence of arrows between neighbouring institutions. We emphasize that such a construction of programming process from metamathematical and also mathematical point of view is not a mechanical activity. Such program development process needs a deep understanding of the metamathematical methods and mathematical theories, well-founded phantasy which can apply practically the whole mathematics in answering real life human questions. The programming is an art as stated Professor Knuth.

4. Conclusion

It is possible before outline metamathematics of computer programming to formulate not only an adequate axiomatic set theory, but also many subtheories of the set theory which can be needed in answering real life questions. A successful example of it is the well-known constraint programming [11]. But we intend to generalize it for every possible mathematical theory and their unions in the framework of an axiomatic set theory.

Because we are unsatisfied with the situation of the world's society, we should like to formulate as a first mathematical theory the theory of games.

Scilicet we would like by computers to help the rational behaviour of persons, families and communities in the worldwide society.

References

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

[2] **Adámek J. and Trnková V.,** *Automata and algebras in categories*, Kluwer, Dordrecht, 1990.

[3] **Cloksin W.F. and Mellish C.S.,** *Programming in Prolog*, Springer, 1987.

[4] **CoFl Task Group on Language Design,** CASL - The CoFl algebraic specification language - Summary, www.brics.dk/Projects/CoFl/Documents/CASL/Summary, 1999.

[5] **Hoffmann M. and Krieg-Brückner B.,** *PROgram Development by SPECification and TRAnsformation: Methodology - Language Family - System*, Springer, *LNCS* **680**, 1993.

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

[7] *The programming language Ada. Reference Manual*, eds. G.Goos and J.Hartmanis, Springer, *LNCS* **155**, 1983.

[8] **Guttag J.V. et al.,** *Larch: Languages and tools for formal specification*, Texts and Monographs in Computer Science, Springer, 1993.

[9] **Hudak P. et al.,** Report on the programming language Haskell, *SIGPLAN Notices*, **27** (5) (1992).

[10] **Messeguer J.,** General logic, *Proc. Logic Colloquium 1987*, North Holland, 1989, 275-329.

[11] **Rossi F.,** Constraint logic programming, *Proc. ERCIM/Compulog Net workshop on constraints*, Springer, *LNAI* **1865**, 2000.

[12] **Schröder L.,** Categories: a free tour, *Categorical perspectives*, eds. A.Melton and J.Koslowski, Birkhäuser, Basel, 2001, 1-27.

[13] **Vopěnka P.,** *Mathematics in the alternative set theory*, Teubner, Leipzig, 1979.

[14] **Wirth N.,** *Data structuress + Algorithms = Programs*, Prentice Hall, Englewood Cliffs, 1975.

[15] **Wulf W., London R.L. and Shaw M.,** An introduction to the construction and verification of ALPHARD programs, *IEEE Trans. Software Eng.,* **2** (1976), 253-265.

(Received June 10, 2002)

V. Novitzká
Technical University
Košice, Slovakia

B. Novitzky
University of P.J. Šafárik
Košice, Slovakia