MATHEMATICAL LANGUAGE IN PROGRAMMING

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

SUMMARY

In our paper we present a dealing with a mathematical language as a basis for formulating different kinds of specifications for a general problem solving by computers. We concentrate on the mathematical basis of data abstractions as program units with mathematically defined meanings and we concern with the development of programs from requirements to results.

Keywords: *data abstraction, specification, models, category, algorithm, institution*

1. INTRODUCTION

The computers of von Neumann type are able to solve problems not only by methods of 'number crunching'. We know that there is computer music, computer graphic, etc. Computers are able to solve theoretical problems which can be described by mathematics and they can also solve practical problems of the fields of mathematical economy, theory of games, econometrics, operation research and many others. These facts inspired us to an intention to solve by computers such aesthetical theoretical and practical problems that are characterizable by mathematics.

In this paper we firstly describe a mathematical language. Then we characterize data abstractions as reusable programming units defined upper mathematical theories, so that they have strong connection to these theories and so that the program development process from requirements to results can be characterized by mappings between such mathematical entities that contain all information about the underlying mathematical theory and the syntax and semantics of data abstractions.

2. MATHEMATICAL LANGUAGE

At the beginning we recall the language of the classical first-order predicate logic [1] and we extend it by new symbols: we enable variables of different *sorts*, and we introduce function symbols of the form

$$f:(s_1,\ldots,s_n)\rightarrow s$$

with their profiles, where $s_1,\ldots,s_n$, s are (not necessarily different) sorts. In such a language we can formulate (true or false) sentences, i.e. closed formulae, and from them we can prove truthness of another sentences. Such extended language of the first-order predicate logic we denote by **L**.

In the language **L** we can formulate axioms of a set theory, for which we know that they are consistent and complete. These two properties of the axiomatic set theory are very important in programming, because the problems that we intend to solve by computers can be formulated in possibly dangerous manner.

Moreover, in the later phase of the program development process we need to extend the language **L** by new notions of algorithms and algorithmic formulae to the language **L'** of *algorithmic logic*. We show this extension in Section 5 of this paper. In algorithmic logic we often need to examine the complexity of algorithms really solving the considered problems by computing values of those functions that correspond with function symbols in signatures of data abstractions and that can be examined in models of them.

3. SPECIFICATIONS UPPER SUBTHEORIES

It is well-known in mathematics that the whole poor and applicable theories can be formulated as conservative extensions of a good axiomatic set theory. In this paper we assume that we already have such axiomatic set theory and a conservative extension of it that is a subtheory of the axiomatic set theory.

Following Niklaus Wirth [2], we can formulate that 'problem specification = data structures + algorithms'. We start the program development process with formulating *scientific presuppositions* of solved problem and *goals* of this problem solving as a *solution*. The presuppositions and solution together we call the *requirements specification* of the problem.

From Software Engineering we know that we can formulate a requirements specification as data abstractions written in some appropriate specification language. The *syntax* of a data abstraction is an *algebraic many-sorted signature*

$$\Sigma = (S, \Omega, Pred)$$

that is already formulated in a very simple mathematical structure in the framework of our subtheory. In the following text we use for simplicity only the word signature for the concept of many-sorted signature. A signature Σ contains the realizations of sorts symbols as a set S, of function symbols as a set Ω and of predicate symbols as a set *Pred* from the logic. Elements of the set *Pred* are interpreted so that the predicate symbols are contained in closed formulae of the subtheory upper which the signature is defined and those closed formulae are pseudo-axioms and/or theorems of this subtheory.

To a signature Σ as a syntax of a data abstraction we firstly construct many-sorted *Σ-algebras* of the form

$$A = (A_S , \Omega_A),$$

where A_S is an *S*-sorted set of *carrier sets* A_s , for every sort $s \in S$, and Ω_A is a set of *functions*

$$f_A: A_{s1} \times \ldots A_{sn} \rightarrow A_s ,$$

for every function symbol $f:(s_1,\ldots,s_n) \rightarrow s$ from Ω.

We are interested in such Σ-algebras, in which are also pseudo-axioms and/or theorems of the subtheory that solve as a basis for proving new sentences about the properties of functions defined on carrier sets. Such many-sorted Σ-algebras we call *Σ-models*. They constitute the *semantics* of the considered data abstraction. We note here, that instead of saying '*a* is an element of a carrier set A_s ', we can use '*a inhabits* in a type A_s '. In such a manner we found the closed relation between many-sorted algebras and *type theory* [3].

From sentences, signatures and models we can construct in unique manner the *institution* that characterize the whole specification. We can consider that function symbols in signatures and functions in the corresponding models are actually a simple or more elaborated form of *algorithms.* As we follows by *arrows* from the requirements specification institution to the *program specification* institution, in which models already contain formulae of algorithmic logic, we are working out a mathematically proved program in some appropriate programming language. We deal with institutions and institution arrows in Section 6 of this paper. We suppose also that the underlying operating system of the computer contains an appropriate program development system that correctly creates *executable specifications* from program specifications and finally the *result specification.* The meaning of these two kinds of specifications are models created by methods of automata theory.

4. DATA STRUCTURES

In the previous section we have established the relation between carrier sets in models of data abstractions and type theory. Types determine data structures in the sense, that all elements of data structures and relations between them are actually determined by the properties of their types. Because category theory deals with mathematical structures and morphisms between them, we use it for formulating concepts and properties of data structures.

An *abstract category* ***A*** consists of ***A***-objects creating a class, of ***A***-morphisms between objects containing identities for every ***A***-object, and of associative composition of ***A***-morphisms. A *functor*

$$F: \boldsymbol{A} \rightarrow \boldsymbol{B}$$

is a morphism from the category ***A*** to the category ***B*** that assigns to every ***A***-object a ***B***-object and to every ***A***-morphism a ***B***-morphism, such that it preserves composition of morphisms and identities.

We can define concrete categories that retain more information about their objects and morphisms. A *concrete category* over a *base category* ***B*** is a pair *(**A**, U)*, where ***A*** is a (total) category and $U: \boldsymbol{A} \rightarrow \boldsymbol{B}$ is a forgetful functor (it forgets some information from ***A***). Detailed definitions of these concepts can be found in [4].

Three basic kinds of type theory, i.e. *simple*, *dependent* and *polymorphic* type theories [5], differ in the forms of indexing types. Simple type theory uses no indexing, dependent type theory uses indexing by term variables and polymorphic type theory uses indexing by type variables.

In our approach we introduce indexing of categories by so-called *fibration* [6]. Putting a fibration on the top of the categorical structure corresponding to type theory, we can put together complicated structures in a modular way. In indexing of categories we use the approach of *display indexing*. Let *X* be a class of sets and *I* be an index set. We define a function $d: X \rightarrow I$, so that the elements (sets) *x* appear as *fibres over i*:

$$d^{-1}(i) = \{ x \in X \mid d(x) = i \}$$

for every $x \in X$ and $i \in I$. The fibres $d^{-1}(i)$ are necessarily disjoint. We say that *d displays X*, and that *X is over I.*

Now we define fibration over categories as follows.

Definition 1: Let *(**A**, U)* be a concrete category over ***B***. The functor $U: \boldsymbol{A} \rightarrow \boldsymbol{B}$ can be seen as a display class

$$(U : \boldsymbol{A} \rightarrow \boldsymbol{B})$$

of categories. For a ***B***-object *I*, the *fibre category*

$$\boldsymbol{A}_I = U^{-1}(I)$$

over I is the category whose objects are ***A***-objects *A* with $U(A) = I;$ and whose morphisms are ***A***-morphisms $f: A \rightarrow A'$ with $U(f) = id_I$, i.e. identities on *I* in ***B***.

□

Example 1: As an example we can construct the category ***Sign*** of signatures and its fibration.

Sign-objects are signatures $\Sigma = (S, \Omega, Pred)$ and ***Sign***-morphisms are signature morphisms $\sigma : \Sigma \rightarrow$

Σ', which satisfy the properties of associativity and the existence of identities.

We can construct a forgetful functor

$$U\colon \textbf{\textit{Sign}} \rightarrow \textbf{\textit{Set}}$$

From this category of signatures to the category of sets (that contains sets as objects and functions between them as morphisms) as follows:

- functor U assigns to every signature Σ its underlying set of sorts (types) S; and
- functor U assigns to every signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ the function $u\colon S \rightarrow S'$ between the corresponding set of sorts.

We construct the category *Class(**Set**),* whose objects are classes of sets $(X_i)_{i\in I}$ and morphisms are pairs (u, f), where f is a class of morphisms

$$f_i : X_i \rightarrow Y_{u(i)}.$$

The functor $K\colon Class(\textbf{\textit{Set}}) \rightarrow \textbf{\textit{Set}}$ is trivially a fibration.

For every signature $\Sigma = (S, \Omega, Pred)$, the set Ω can be consider as a function

$$\Omega\colon S^* \times S \rightarrow \textbf{\textit{Set}},$$

that yields for every $(n+1)$-tuple $(s_1,\ldots,s_n, s)$ a set

$$\Omega_{(s1,\ldots,sn)\rightarrow s}$$

of function symbols with the profile $(s_1,\ldots,s_n)\rightarrow s$, and $Pred$ can be considered as a function

$$Pred\colon S^* \rightarrow \textbf{\textit{Set}},$$

that yields for every n-tuple $(s_1,\ldots,s_n)$ a set of predicate symbols with the arity $(s_1,\ldots,s_n)$.

Then we can construct the functor $G\colon \textbf{\textit{Set}} \rightarrow \textbf{\textit{Set}}$, which assigns to every set S of sorts from Σ a set

$$(S^* \times S) + S^*,$$

where '+' denotes disjoint union of sets.

From the change-of-base theorem [5] and from the pullback diagram in Figure 1 we obtain that $U : \textbf{\textit{Sign}} \rightarrow \textbf{\textit{Set}}$ is fibration functor.

$$\begin{tikzcd}
\textbf{\textit{Sign}} \arrow[r] \arrow[d, "U"'] \arrow[dr, phantom, "\lrcorner", very near start] & Class(\textbf{\textit{Set}}) \arrow[d, "K"] \\
\textbf{\textit{Set}} \arrow[r, "G"'] & \textbf{\textit{Set}}
\end{tikzcd}$$

Fig. 1 Pullback of ***Sign***

Fibration ***Sign*** $\rightarrow$ ***Set*** implicitly defines also the type system of a model that will be the semantics of signatures. This fibration is also useful because it enables construction of function values of a concrete types (by using algorithmic logic, see Section 5), i.e. values which inhabit in carrier sets indexed by sorts.

5. ALGORITHMS

In requirements specifications we have in their syntax function symbols and in their semantics functions. These function symbols and functions shortly express an algorithm that construct the values of the functions. The construction of function values needs more as allows the first-order predicate logic. Some kind of the intuitionistic logic is needed. We make the first step to the extension of the first-order predicate logic to an intuitionistic logic that enables on the basis of proofs in models also the so-called *deliverables*, i.e. preconditions, expressions, postconditions and so also the final result of the basis of proved annotation.

We extend our logical language **L** to the language **L'** of *algorithmic logic.* We add to symbols of the language **L** new symbols called *algorithms* as follows:

- An algorithm is every expression of the form

$$x{:=}\, t \quad \text{or} \quad q{:=}\, \psi$$

where x is a variable and t is a term of the same sort s ; q is a proposition (a predicate symbol with zero arity) and ψ is an open formula. Such algorithm we call *assignment.*

- Let P and P' be algorithms. An algorithm is

i) every expression of the form

$$\textbf{\textit{if}}\ \psi\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ P'\ \textbf{\textit{fi}}$$

that we call *branching* between P and P';

ii) every expression of the form

$$\textbf{\textit{while}}\ \psi\ \textbf{\textit{do}}\ P\ \textbf{\textit{od}}$$

that we call *iteration* of P;

iii) every expression of the form

$$\textbf{\textit{begin}}\ P;\ P'\ \ \textbf{\textit{end}}$$

that we call *composition* of P and P'.

We can recognize these algorithms as special function symbols on the set **P** of all algorithms. The iteration can be recognized as the unary function symbol '$*_\psi$', composition and branching as binary function symbols 'o' and 'if_ψ', respectively, with the function profiles as follows:

$$*_{\psi}: \mathbf{P} \rightarrow \mathbf{P}$$
$$o : (\mathbf{P}, \mathbf{P}) \rightarrow \mathbf{P}$$
$$if_{\psi} : (\mathbf{P}, \mathbf{P}) \rightarrow \mathbf{P}.$$

Now we extend the language also by new kind of formulae, so-called *algorithmic formulae.* Assume that P is an algorithm and α is a formula. An algorithmic formula can be of one of the following forms:

- as $P\alpha$, i.e. values of variables occurring in P are assigned to the variables occurring in the formula α;
- as $\cup P\alpha$, where the symbol '$\cup$' acts as the *existential iteration operator*;
- as $\cap P\alpha$, where the symbol '$\cap$' acts as the *universal existential operator.*

The language **L** extended with algorithms and algorithmic formulae is the language of algorithmic logic and we denote it by **L'**.

Example 2: Consider a simple example of an algorithm and two algorithmic formulae.

Let x, y, z, i be variables of the same sort s; '+', '-', '.' two binary predicate symbols:

$$+: (s, s) \rightarrow s$$
$$-: (s, s) \rightarrow s$$
$$. : (s, s) \rightarrow s$$

a and b two function symbols with no arguments, i.e. constants:

$a: s$ and $b: s$

and '≤', '<' be two binary predicate symbols

$\leq : (s, s)$ and $< : (s, s)$.

Then P is an algorithm:

```
P: begin z:= z; i:= a;
    while y ≤ z do
            z:= z – y;
            i:= i + b
        od;
  end;
```

Algorithmic formulae can be as following:

$$P(x = ((i.y) + z) \wedge (a \leq i)))$$
and
$$(\exists y)((x{:=}\ y) \cup (x{:=}\ x+b)(z \leq x))$$

□

From this example we can say that annotated program (with special comments of the form of formulae) can be treated in algorithmic logic as algorithmic formulae.

The algorithmic construction is an extension of the proof of sentences in the models of signatures and so as semantics of data abstractions. This extended model enables the construction of already normal programs in some appropriate programming languages according to the algorithmic constructions in models.

6. INSTITUTIONS

The concept of institution enables us to describe the whole specification in unique manner. An institution is a construction that put together all important information about specification: its signatures, i.e. syntax of data abstractions; its models; and axioms and/or theorems of the conservative extension of the axiomatic set theory upper which the specification is considered.

Formally we define an institution as follows.

Definition 2: An *institution of a specification*

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

consists of

- a category ***Sign*** of signatures of data abstractions from the specification;

- a functor

$$Sen: \boldsymbol{Sign} \rightarrow \boldsymbol{Set}$$

providing to every signature Σ a set of Σ-sentences, i.e. axioms and/or theorems of the considered subtheory; and to every signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ a mapping translating Σ-sentences to Σ'-sentences;

- a functor

$$Mod: \boldsymbol{Sign}^{op} \rightarrow \boldsymbol{Cat}$$

from the dual category of signatures to the category of small categories that provides for every signature Σ from ***Sign*** a category $Mod(\Sigma)$ of Σ-models; and to every signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ a functor $Mod(\sigma) : Mod(\Sigma') \rightarrow Mod(\Sigma)$;

Signature morphisms have to preserve validity of sentences in corresponding models.

□

In the program development process we need a well-defined mapping from one kind of a specification to the next kind, e.g. from requirements specification to the corresponding program specification, etc. Because any specification is uniquely determined by an institution, it is enough to construct a morphism between institutions, i.e. some *arrow* from one institution to the other one. There are several possibilities of constructing such arrows. Because we assume that we construct from the 'more poor' institution a 'richer one', which has more insformation about specification, we choose the concept of institution representation [7].

Let $\boldsymbol{I} = (\boldsymbol{Sign}, Sen, Mod)$ and $\boldsymbol{I'} = (\boldsymbol{Sign'}, Sen', Mod')$ be such institutions. An *institution representation*

$$\rho : \boldsymbol{I} \rightarrow \boldsymbol{I'}$$

consists of

- a functor

$$\rho^{Sig}: \boldsymbol{Sign} \rightarrow \boldsymbol{Sign}\ ;$$

- a natural transformation (i.e. a morphism between functors)

$$\rho^{Mod}: (\rho^{Sig})^{op} \circ Mod' \rightarrow Mod,$$

i.e. a class of functions $(\rho_{\Sigma}^{Mod})_{\Sigma\in \boldsymbol{Sign}}$, such that the diagram in Figure 2 commutes:

$$\begin{tikzcd}
\Sigma_1 \arrow[d, "\sigma"'] & Mod(\Sigma_1) & Mod'(\rho^{Sig}(\Sigma_1)) \arrow[l, "\rho_{\Sigma 1}^{Mod}"'] \\
\Sigma_2 & Mod(\Sigma_2) \arrow[u, "Mod(\sigma)"] & Mod'(\rho^{Sig}(\Sigma_2)) \arrow[l, "\rho_{\Sigma 2}^{Mod}"] \arrow[u, "Mod(\sigma')"']
\end{tikzcd}$$

Fig. 2 Natural transformation ρ^{Mod}

- a natural transformation

$$\rho^{Sen}: Sen \rightarrow \rho^{Sig} \circ Sen',$$

i.e. a class of functions $(\rho_{\Sigma}^{Sen})_{\Sigma\in \boldsymbol{Sign}}$, such that the diagram in Figure 3 commutes and preserve the validity of sentences in corresponding models.

$$\begin{tikzcd}
\Sigma_1 \arrow[d, "\sigma"'] & Sen(\Sigma_1) \arrow[r, "\rho_{\Sigma 1}^{Sen}"] \arrow[d, "Sen(\sigma)"'] & Sen'(\rho^{Sig}(\Sigma_1)) \arrow[d, "Sen'(\rho^{Sig}(\sigma)"] \\
\Sigma_2 & Sen(\Sigma_2) \arrow[r, "\rho_{\Sigma 2}^{Sen}"'] & Sen'(\rho^{Sig}(\Sigma_2))
\end{tikzcd}$$

Fig. 3 Natural transformation ρ^{Sen}

In the framework of an institution arrow the construction of the appropriate mappings is not trivial. For example, it is not trivial to construct such mappings if during the program development process we map an institution constructed upper one subtheory onto an another institution constructed upper another subtheory. In such a case the mappings should respect the rather different axioms and/or theorems constituting the sentences of the institutions and their role in two models.

Moreover, if the second institution is an institution in which models contain deductions in the framework of algorithmic logic, the mappings has to respect the relation between proofs in two different logics. In algorithmic logic the proofs are more difficult and such a more difficult model is actually a target model of the mapping which has to be constructed so that it simultaneously proves also correctness of the arrow between institutions.

7. CONCLUSION

In our paper we have tried to show that the programming process from requirements to results (in this short presentation only from requirements to program) is intellectually a non-trivial process. This process is highly rationalistic and strictly philosophically founded scientific process.

REFERENCES

[1] Sochor, A.: Klasická matematická logika, Karolinum, Praha 2001
[2] Wirth, N: Data Structures + Algorithms = Programs, Prentice-Hall, 1975
[3] Pierce, B. C: Types and Programming Languages, MIT Press, Cambridge, 2002
[4] Adámek, J. et al.: Abstract and Concrete Categories, Wiley & Sons, New York, 1990
[5] Jacobs, B: Categorical Logic and Type Theory, Elsevier, Amsterdam, 1999
[6] Grothendieck, A.: Catégories, fibrées et descente, Revétement Etales Groupe Fondamental, No 224, Springer, 1970, pp.145-194
[7] Novitzká, V.: O teórii korektného programovania, Academic Press Elfa, Košice, 2003

BIOGRAPHY

Valerie Novitzká defended her PhD. Thesis “On formal semantics of Anna” in 1989 in Budapest at Hungarian Academy of Sciences. She is working as lecturer at the Department of Computers and Informatics. Her scientific research is focusing on the theoretical foundations of programming and questions relating with specifications, type theory, program correctness and semantics.