LOGICAL REASONING ABOUT PROGRAMMING OF MATHEMATICAL MACHINES

Valerie NOVITZKÁ
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University Košice, Letná 9, 042 10 Košice, Slovak Republic, E-mail: Valerie.Novitzka@tuke.sk

SUMMARY
We always start the solving of a problem with the formulation of its theoretical foundations. If we would like to use mathematical machines (computers) in problem solving, we need to formalize its theoretical foundations as logical reasoning because the programs should really prove the correctness of their results. In our paper we present central ideas of our approach regarding programming as logical reasoning. Our first idea is that the theory in which we reason is the type theory starting with basic types. Our second idea is that the running program is actually a proof in the theory above formulated as the intuitionistic linear version of Gentzen's calculus. We show that such a synthesis of categorical and linear logic forms a theoretical foundations of programming for mathematical machines.

Keywords**:** *category theory, topos, categorical logic, type theory, linear logic*

1. INTRODUCTION

One of the famous computer scientist, Niklaus Wirth, defined programs as data structures and algorithms. In his book [1] he thoroughly explained how to develop middlesize Pascal programs, how to define types, declare data structures, formulate procedures, functions and main programs in the rational manner. It is true that Pascal is until now the most rational procedural programming language. His approach has minimally two drawbacks: the first is that the program cannot check the malfunctionality of hardware; the second, that he does not give the answer to the question how exactly develop correct programs in which the results are really proved.

Traditional software engineering approach to solve the second drawback is the following: a client formulates his requirements specification and a programmer derives from it a program in whatever programming language, he edits (normally or structurally) text, compiles it, uses librarian, linkage editor, loader, routines of operating system, executes his program and gets some results that are not mathematically proved. Every possible form of testing whether the program does the required actions cannot be concerned as proof neither in Aristotelian notion of a logical proof or in the sense of John Stuart Mill's idea of inductive logic.

We attempt to formulate a theoretical foundations of program development. In such a manner program results after execution have to be formally proved using mathematical ideas, lemmas and theorems. We be aware of the meaning of czech computer scientist Antonín Svoboda that computers are mathematical machines which can realize mathematical forms of human thinking. We try to find such mathematical disciplines which can describe in formally exact way the whole process of development and execution of programs. We begin with the question: what actually a program is?; i.e. what does a program perform? Our analysis leads to the answer that a program solves human rational problems, i.e. really scientific problems. Solving of such problems is possible only in a framework of a mathematical theory. We find a discipline, categorical logic, formulated in the last decade [2], which is able to describe by the help of types, terms, morphisms and functors the whole process of program development starting from categories of basic types to a total category. Indeed, categorical logic can exactly describe problem solving process in the framework of program development process in mathematically constructive manner .

Total category, the intermediate result of this process can be mapped to the categorical semantics of linear logic, a new approach to mathematical logic, which respects actions of mathematical machines in very disciplinary way.

In this paper we should like to explain the startpoint of program development in categorical logic, i.e. basic types, and basic notions of semantics of linear logic that can be mapped to some programming language. Our approach is practically founded in mathematics, logics and type theory and gives new ideas not only for constructing new proof assistants but also to the new development of hardware by extending it by new processors of basic types. This paper is the starting one which will be followed by more detail explanation of further aspects of categorical logic and linear logic, i.e. of exact reasoning proving the results of programs. We hope that these correct programs could show hardware errors and so increase the reliability of program execution.

2. BASIC CONCEPTS FOR PROBLEM SOLVING

From the disscussion in previous section it follows that the resulting data structure of executed program have to give proved answers to the questions formulated at the beginning of program development process. We can achieve these only in the case when the program itself represents such problem solving process which is formulated in a well-formed and general mathematical theory. After a thorough foundation of the acceptable logical reasoning leading to the proved result we decided to start our reasoning with the most general mathematical theory, i.e. with category theory .

We will shortly introduce here only those notions from category theory that serve for our purposes. A category ***C*** consists of two collections: a collection of objects *A, B, ...;* and a collection of arrows (or morphisms) denoted *f, g,* .... For and arrow $f\colon A \to B$ the objects *A* and *B* are the domain and codomain, respectively. A homset *Hom(A, B)* is the set of all arrows with domain *A* and codomain *B* in ***C***. The arrows $f\colon A \to B$ and $g\colon B \to C$ are composable, its composition $g \circ f = g f : A \to C$ is associative in ***C*** and every object *A* has identity arrow $id_A\colon A \to A$. We say that a category ***C*** has terminal object **1** (initial object **0**) if for every category object *A* there exists unique arrow $A \to \mathbf{1}$ ($\mathbf{0} \to A$).

Categories as other kinds of mathematical structures come equipped with the notion of homomorphism between categories called functor. A functor $F\colon \boldsymbol{C} \to \boldsymbol{D}$ from a category ***C*** to the category ***D*** takes the objects of ***C*** to the objects of ***D*** and the arrows of ***C*** to the arrows of ***D*** preserving domains, codomains, identities and composition. If $F,G\colon \boldsymbol{C} \to \boldsymbol{D}$ are functors with the common domain and codomain, a natural transformation $\mu\colon F \to G$ is a family of arrows $\mu A\colon FA \to GA$ for every object *A* of the category ***C***, such that for every arrow $f\colon A \to B$ in ***C***

$$Gf \circ \mu A = \mu .$$

We note here that the objects of a category are something so general that in a category we can only distinghish one object from other ones. In general category theory we do not suppose anything more about the nature of category objects. But of course, these objects may be sets in the sense of an axiomatic set theory. Therefore we can denote by ***Set*** the category of sets as objects and functions between them as morphisms.

For a category ***C*** and its fixed object *A* we define hom functor $Hom(A, -)\colon \boldsymbol{C} \to \boldsymbol{Set}$, which assigns to every object *B* from the category ***C*** the homset *Hom(A ,B)* of all arrows from the object *A* to *B*, and to every arrow $f\colon B \to C$ a function

$$Hom(A, f)\colon Hom(A, B) \to Hom(A, C).$$

One of the important perceptions of category theory is that an arrow $x\colon T \to A$ in a category ***C*** can be regarded as an element of *A* over *T*. An object *T* is called the domain of variation of *x* and *x* is called a (variable) element of the object *A*. This method enables a generalization of the set-theoretic membership relation. It is clear that any object *A* of a category ***C*** has at least one element id_A , its generic element.

We denote by *Func(**C**, **Set**)* the category of functors $F\colon \boldsymbol{C} \to \boldsymbol{Set}$ from the category ***C*** to the category ***Set*** as objects and natural transformations between them as arrows. Similarly as above, elements of a functor *F* are natural transformations from objects of functor category, i.e. the functors, into the functor *F*. From Yoneda lemma [3] it follows: if an element *u* of a functor *F* over the hom functor *Hom(A, −)* is a natural *A* - isomorphism, $u \in FA$, then this unique *u* is called the universal element for functor *F*. A functor *F* that has the universal element is representable functor. A functor $Sub\colon \boldsymbol{C} \to \boldsymbol{Set}$ is a subobject functor if it assigns to every object *A* in ***C*** the set of subobjects of *A*.

Category theory expresses equations by means of commutative diagrams. A diagram *D* in a category ***C*** is a graph homomorphism $D\colon \boldsymbol{I} \to \boldsymbol{C}$, where ***I*** is the index (shape) graph of the diagram *D*. A diagram *D* is commutative, if all paths from an object *A* to the object *B* in diagram constructed as compositions of corresponding arrows are equal, e.g. the diagram in Fig. 1. commutes

$$\begin{tikzcd}
 & \bullet \arrow[r, "f_2"] & \bullet & \cdots & \bullet \arrow[r, "f_{n-1}"] & \bullet \arrow[dr, "f_n"] & \\
A \arrow[ur, "f_1"] \arrow[dr, "g_1"'] & & & & & & B \\
 & \bullet \arrow[r, "g_2"'] & \bullet & \cdots & \bullet \arrow[r, "g_{m-1}"'] & \bullet \arrow[ur, "g_m"'] &
\end{tikzcd}$$

Fig. 1 Commutative diagram in category

and expresses the following equation:

$$f_n f_{n-1} \cdots f_2 f_1 = g_m\, g_{m-1} \cdots g_2\, g_1 .$$

A commutative cone α with vertex *V* over a diagram *D* is an element of *D* over a constant diagram *V*, as in Fig. 2.

$$\begin{tikzcd}
 & & V \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
{} & \cdots D_{i-1} \arrow[rr, shift left] & \alpha_i & D_i \cdots \arrow[ll, shift left] & {}
\end{tikzcd}$$

Fig. 2 Commutative cone

A functor $Cone\,(-, D)\colon \boldsymbol{C} \to \boldsymbol{Set}$ assigns to an object *V* of the category ***C*** the set *Cone(V, D)* of commu- tative cones with vertex *V* over the diagram *D*. A universal element of the functor *Cone(−, D),* if exists, is a limit of the diagram *D*. A category ***C*** has finite limits if every finite diagram D in $\boldsymbol{C}$ has a limit.

If A and B are objects of a category $\boldsymbol{C}$, then $A \times B$ is the product object together with two arrows (projections):

$$\pi_1 : A \times B \rightarrow A \qquad \text{and} \qquad \pi_2 : A \times B \rightarrow B,$$

such that for any object C and arrows $f: C \rightarrow A$ and $h: C \rightarrow B$ there exists unique arrow $h: C \rightarrow A \times B$ with

$$\pi_1\ h = f \quad \text{and} \quad \pi_2\ h = g$$

as in Fig. 3.

$$\begin{tikzcd}
A \times B \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & B \arrow[d, "g"] \\
A & C \arrow[l, "f"] \arrow[ul, "h"']
\end{tikzcd}$$

Fig. 3 Cartesian product in category

To generalize cartesian product of objects of a category $\boldsymbol{C}$ we have to introduce the concept of indexing. Let $\boldsymbol{B}$ be a base category with the index sets $I, J, \dots$ as objects and functions $u: I \rightarrow J$ between them as arrows. We define two way of indexing of objects in a category $\boldsymbol{C}$:

1. pointwise indexing by a functor $\boldsymbol{B} \rightarrow \boldsymbol{C}$, such that it assigns to every index set object I a family $(A_i)_{i \in I}$ of objects from $\boldsymbol{C}$;
2. display indexing by a functor $\boldsymbol{C} \rightarrow \boldsymbol{B}$ which assigns to a subcategory of $\boldsymbol{C}$ the object I indexing it.

Now we can say that a category $\boldsymbol{C}$ has (finite) products if any (finite) indexed family of objects in $\boldsymbol{C}$ has a product. A category $\boldsymbol{C}$ is cartesian closed category (ccc) if the following conditions hold:

1. $\boldsymbol{C}$ has a terminal object $\boldsymbol{1}$;
2. $\boldsymbol{C}$ has finite products;
3. for any pair of objects A and B in $\boldsymbol{C}$ there is an exponential object B^A such that for every object C in $\boldsymbol{C}$ the following homsets are isomorphic

$$Hom(C \times A, B) \cong Hom(C, B^A).$$

A topos is a special kind of category defined by axioms saying roughly that certain constructions one can make with sets can be done in a category. A topos is a category $\boldsymbol{E}$ which satisfies the following properties:

1. it is ccc,
2. it has finite limits;
3. it has representable subobject functor.

Moreover, if we want a topos to be a generalized mathematical theory, we suppose that a set of hypotheses or axioms are formulated in predicate logic. They implicitly define some kind of structure of objects and some properties of morphisms in the category $\boldsymbol{E}$. A topos is really a structure of a general theory defined by axioms formulated possibly in higher-order logic. An elementary topos is such one whose axioms are formulated in the first-order logic, i.e. as it was mentioned above by defining element, elementary topos is generalized axiomatic set theory.

In elementary topos we can define a mathematical structure on every its object in the sense of [4]. Such a structure is an ordered sequence

$$U = (M, R_1, \dots, R_n, F_1, \dots F_m, \{c_i\}_{i \in I}),$$

where M is a non-empty set, $R_1, \dots R_n$ are relations on M, $F_1, \dots F_m$ are functions on M and c_i are elements (constants) of M. Of course, the properties of mathematical entities of such a structure are also determined by axioms that can be regarded as a conservative extension of axioms of set theory.

An elementary topos whose objects are some distinguished structures we regard as a category of basic types. Categories of basic types are the starting point for constructing more complex types via functors representing also the type-theoretical constructors for simple types ($\times$, $\rightarrow$, $+$), dependent types, inductive types, recursive types [5] (by recursive morphisms and functors) and polymorphic types (allowing type variables) as in [2] and of course the morphisms between such constructed objects.

So, we have formulated a way how to construct arbitrary data structures according to the types and a mathematical discipline of derivation data structures from simple ones starting with basic types. We note that like the floating point processor, some of denoted basic types (which can be not only trivial) would need their own processors in hardware of mathematical machines of future.

Constructive process of mathematically proved and uniquely typed data structures finishes with a total category contaning proofs and results of our problem solving process. We note that it is possible to think about such a constructive approach to problem solving on account of the continually evolving discipline called categorical logic.

3. BASIC CONCEPTS FOR PROGRAMMING

After the basic theoretical reasoning which will finish by the construction of a total category containing the result of problem solving or answering the interested questions by a didaction based on a well-formed theories, we follow by construction of a program that realizes the categorical logical reasoning in a form of linear logic. But we mention that in the last two decades we are more and more convinced of the basic importance of the semantics, because from the semantical framework we can already generate an understandable syntax of some text characterizing logical reasoning. This circumstance was comprehended already in the framework of classical mathematical logic, where the most important questions of the consistency, completeness and independence of axioms of the theories must be proved in mathematical structures or models. We do not divide the syntactical writing of a logical reasoning as a proof from the structures, in which the symbols of logic get their senses and denotations, i.e. from the appropriate structures. This is why we will map firstly the total category to the categorical semantics of linear logic and generate the syntactical aspects and visible sequent calculus by a proved manner too from this semantics.

Linear logic was introduced by J.Y.Girard [6] and its simplicity and elegance makes it suitable also for reasoning about programming mathematical machines. To start to work on linear logic reasoning we need to construct a functor that maps the total category to the appropriate category forming the categorical semantics of linear logic.

In the next we formulate the categorical semantics of linear logic for which we use the following concepts. A commutative quantale is a structure of the form

$$(Q, \leq, \circ, 1, 0, \top),$$

where $(Q, \leq, 0, \top)$ is a complete lattice and $(Q, \circ, 1)$ is a commutative monoid. The symbol 'o' denotes a monoid multiplicative operation with the neutral element 1 such that it distributes over suprema:

$$a \circ (\bigvee_{i\in S} b_i) = \bigvee_{i\in S} (a \circ b),$$

where $S \subseteq Q$. The multiplication in such monoid is called fusion. The monoid forms a locale. For quantales *Q and Q'* a quantale homomorphism is a mapping $q: Q \to Q'$, which preserves fusion, its neutral element 1, and suprema. We can construct the category ***CQuant*** of quantales as objects and quantale homomorphisms between them as arrows. It is clear that for every quantale Q there is identity homomorphism $id_Q: Q \to Q$, the quantale homomorphisms are composable and this composition is associative. Therefore we can say that ***CQuant*** is a category. For any quantale Q we introduce residuation as a binary operation defined as follows: for any $a,b \in Q$

$$a \multimap b = \bigvee\{x \mid x \circ a \leq b\}.$$

Residuation operation corresponds with interesting aspects of a connective of linear logic as we shaw later.

A monoidal category $\boldsymbol{C} = (\boldsymbol{C}, \otimes, I, a, l, r)$ consists of

1. a category $\boldsymbol{C}$;
2. a tensor functor $\otimes : \boldsymbol{C} \times \boldsymbol{C} \to \boldsymbol{C}$;
3. natural isomorphisms a, l, r

$$a_{X,Y,Z} : (X\otimes Y)\otimes Z \to X\otimes (Y\otimes Z)$$
$$l_X : I\otimes X \to X$$
$$r_X : X\otimes I \to X,$$

where X, Y, Z are objects of the category $\boldsymbol{C}$. The first isomorphism expresses associativity of tensor functor, the two latter left and right neutral element of it. They have to satisfy the coherence axioms expressed by the following diagrams: pentagon and triangel in Fig. 4 and Fig. 5, respectively.

$$\begin{tikzcd}
((W\otimes X)\otimes Y)\otimes Z \arrow[r, "a\otimes id"] \arrow[d, "a"'] & (W\otimes (X\otimes Y))\otimes Z \arrow[dd, "a"] \\
(W\otimes X)\otimes (Y\otimes Z) \arrow[d, "a"'] & \\
W\otimes (X\otimes(Y\otimes Z)) & W\otimes ((X\otimes Y)\otimes Z) \arrow[l, "id\otimes a"]
\end{tikzcd}$$

Fig. 4 Pentagon – coherence axiom for isomorphism *a*

$$\begin{tikzcd}
(X\otimes I)\otimes Y \arrow[rr, "a"] \arrow[dr, "r\otimes id"'] & & X\otimes (I\otimes Y) \arrow[dl, "id\otimes l"] \\
& X\otimes Y &
\end{tikzcd}$$

Fig. 5 Triangle – coherence axiom for isomorphisms l and *r*

A monoidal category is strict if the isomorphisms *a* , *l* and *r* are identities. To achieve commutativity of tensor product we add to monoidal category a natural isomorphism

$$c_{X,Y} : X\otimes Y \to Y\otimes X,$$

which satisfies coherence axioms in Fig. 6 and Fig. 7, and we call such category symmetrical monoidal category.

For example, if $\boldsymbol{C}$ is a category with finite products, it is easy to show that it is symmetric monoidal category, tensor functor $\otimes$ is here given by cartesian product, I is the terminal object and isomorphisms *a, l, r, c* are given by appropriate combinations of pairing and projections.

$$\begin{tikzcd}
(Y\otimes X)\otimes Z \arrow[d, "a"'] & (X\otimes Y)\otimes Z \arrow[l, "c\otimes id"'] \arrow[d, "a"] \\
Y\otimes(X\otimes Z) \arrow[d, "id\otimes c"'] & X\otimes(Y\otimes Z) \arrow[d, "c"] \\
Y\otimes(Z\otimes X) & (Y\otimes Z)\otimes X \arrow[l, "a"]
\end{tikzcd}$$

Fig. 6 Coherence axiom for *a* and *c* isomorphisms

$$\begin{tikzcd}
& X\otimes Y \arrow[dl, "c"'] \arrow[dr, "id"] & \\
Y\otimes X \arrow[rr, "c"'] & & X\otimes Y
\end{tikzcd}
\qquad
\begin{tikzcd}
I\otimes X \arrow[rr, "c"] \arrow[dr, "l"'] & & X\otimes I \arrow[dl, "r"] \\
& X &
\end{tikzcd}$$

Fig. 7 Coherence axiom for *c, l* and *r* isomorphisms

A symmetric monoidal category $\boldsymbol{C}$ is closed, if for every object X in $\boldsymbol{C}$ the functor $-\otimes X$ has a specified right adjoint, the hom functor $Hom\,(X, -)$

$$-\otimes X \dashv Hom\ (X, -),$$

that is, there exist natural transformations

$$\varepsilon_{X,Y}: Hom(X, Y)\otimes X \to Y \quad and$$

$$\delta_{X,Y}: X \to Hom(Y, X\otimes Y),$$

which satisfiy the triangle identities for an adjunction

$$1 = \varepsilon\,(\delta\otimes 1): X\otimes Y \to Hom(Y, X\otimes Y)\otimes Y \to X\otimes Y$$

and

$$1 = Hom(1, \varepsilon)\ \delta :$$
$$Hom(X,Y) \to Hom(X, Hom(X, Y)\otimes X) \to Hom\ (X, Y).$$

Every quantale Q regarded as a category is strict symmetric monoidal category. So, we constructed strict symmetric monoidal category as unique semantics of the whole linear logic. This categorical semantics gives a possibility to introduce the sequent calculus of linear logic in the category ***CQuant***.

Then we can write down syntax of linear logic defined e.g. in [7] by a sequent calculus. It is not the aim of this short paper to characterize the all aspects of linear syntactical reasoning in the framework of sequent calculus, we only point out that it is proof oriented. Linear reasoning using this calculus may contain very difficult proof nets [8] with possible interactions.

Because linear logic is different from classical logic, but is an extension of it, we illustrate here differences between some logical connectives at least in two cases. Linear logic formulas are actions. In contrast to classical logic where conjunction has a very simple Tarski-Hilbert semantics, the linear logic conjunction is fusion, that is two operands of linear logic conjunction can actually 'annihilate' by the fusion (the notion of annihilation is known from high-energy physics). In linear implication we say that the first operand ontologically causes the result of implication, i.e. the second operand. Such explanation only help our phantasy, the exact semantics we formulated mathematically above. Here we would like to mention that some new research projects about semantics of linear logic [9] may help to conceive the proof nets also as a purposeful finding of conclusion and so combining the ontological causality with the ontological teleology. We try to paraphrase the original Girard's example for linear implication. Assume that the first operand A (action) is the sentence' I spend some amount of money ' and the second operand B is the following one' I get some article'. The linear implication

$$A \multimap B$$

expresses that I spent some amount of money and then I have got some article. But after implication I have not this amount of money. That means the cause of implication is also annihilated after the relization of linear implication. This circumstance has further interesting property of linear reasoning. This reasoning has stages (as it is shown in the framework of Zermelo-Fraenkel set theory formulated in linear logic) in which the following stage rewrites the previous stage. So, the reasoning in linear logic realizes also garbage collection.

Finishing our short description we would like to mention that from the categorical semantics of linear logic we can generate a category for a functional programming language of linear logic by appropriate functor containing also its syntax in ASCII form.

4. CONCLUSION

In this introducing paper we have shown that we can solve scientific problems based theoretically in the framework of categorical logic over basic types. After obtaining the mathematical solution we can construct a linear logic reasoning which represents this solution by proofs in the semantics and syntax of linear logic and so this solution can give in mathematical machine the desired type structure formulated in a theory based on intuitionistic linear logic.

This work was supported by VEGA Grant No.1/2181/05: Mathematical Theory of Programming and Its Application in the Methods of Stochastic Programming

REFERENCES

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

[2] B. Jacobs: *Categorical Logic and Type Theory*, Elsevier, Amsterdam, 1999

[3] M. Barr, Ch. Wells: *Toposes, Triples and Theories*, Springer, 2002

[4] D. van Dalen: *Logic and Structure*, Springer, 1994

[5] A. Eppendahl: *Categories and Types for Axiomatic Domain Theory*, PhD. Thesis, Univ. London, 2003

[6] J.Y.Girard: Linear Logic, *Theoretical Computer Science*, 50, 1987, pp. 1-102

[7] J.Y.Girard: Linear Logic: Its Syntax and Semantics, In: J.Y.Girard, Y.Lafont, and L.Regnier, editors, *Advances in Linear Logic*, Cambridge, 1995, pp. 1-42

[8] Y.Lafont: Interaction Nets, In: *17th Annual Symposium on Principles of Programming Languages*, San Francisco, 1990, pp. 95-108

[9] A.Blass: A game semantics of linear logic, *Annals of Pure and Applied Logic*, 56, 1992, pp. 183-220

BIOGRAPHY

Valerie Novitzka defended her PhD Thesis: On semantics of specification languages at Hungarian Academy of Sciences in 1989. She works at Department of Computers and Informatics from 1998, firstly as Assistent Professor, from 2004 as Associated Professor. Her research areas covers category theory, categorical logic, type theory, classical and linear logic and theoretical foundations of program development.