STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LII**, Number 1, 2007

DEPENDENT TYPES IN MATHEMATICAL THEORY OF PROGRAMMING

VALERIE NOVITZKÁ, ANITA VERBOVÁ

Abstract. In our approach we consider programming as logical reasoning over type theory of a given solved problem. In our paper we follow our work with describing dependent type theory categorically. We introduce dependent types as families of types indexed by terms and we provide rules of dependent type calculus. We describe indexing in terms of special functors, fibrations along display maps over category of type contexts.

1. INTRODUCTION

In our research we consider programming as logical reasoning over type theory of a given solved problem. We built a category of corresponding type theory and logical system over the type theory by fibration, i.e. by a special functor [4] enabling indexing and substitution. In our previous work we presented how to construct in this manner Church's type theory (ChTT) [7] and polymorphic type theory (PTT) [9] categorically and in [8] we built first order logical system over these type theories.

We started our approach with the concept of many-typed signature $\Sigma = (T, F)$ consisting of a finite set T of basic types $\sigma, \tau, \dots$ needed for a solved problem and a finite family F of operations of the form $f : \sigma_1, \dots, \sigma_n \to \tau$. From basic types we can construct Church's types using constructors for product $(\sigma \times \tau)$, coproduct $(\sigma + \tau)$ and function $(\sigma \to \tau)$ types and we defined ChTT by classifying category $Cl(\Sigma)$ over Σ consisting of type contexts (variable declarations) $\Gamma = (v : \sigma_1, \dots, v_n : \sigma_n)$ as category objects and tuples of terms $(t_1, \dots, t_m) : \Gamma \to \Delta$ as category morphisms, where $\Gamma \vdash t_i : \tau_i$ denotes a term t_i of type τ_i for $i = 1, \dots, m$ with free variables declared in Γ.

Then we constructed PTT over higher-order signature $(\bar{\Sigma}, (\Sigma_k))$. PTT enables type variables $\alpha, \beta, \dots$ and kinds $K, L, \dots$ of types that are enclosed in kind signature $\bar{\Sigma} = (\mathcal{K}, \mathcal{F})$ of kinds and functions. For every kind $k \in \mathcal{K}$ a signature Σ_k

Received by the editors: May 25, 2007.

2000 *Mathematics Subject Classification.* 18A15, 18D30.

1998 *CR Categories and Descriptors.* F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic – *λ-calculus and related systems*; G.2.0 [**Mathematics of computing**]: Discrete mathematics – *General* .

consists of $\bar{\Sigma}$-terms $\alpha_1 : K_1, \ldots, \alpha_m : K_m \vdash \sigma : \texttt{Type}$. Then we constructed PTT by split polymorphic fibration

$$\begin{tikzcd} Cl(\bar{\Sigma}, (\Sigma_k)) \arrow[d] \\ Cl(\bar{\Sigma}) \end{tikzcd}$$

with generic object $\texttt{Type}$ in $Cl(\bar{\Sigma})$.

In this paper we follow our approach with defining another types, dependent types that have been frequently used in computer science. There are several other approach to capture type dependency, we mention here only contextual categories [15] and D-categories [1]. We prefer fibrations because they enable to express indexing and substituting by display maps and we investigate how to describe DTT categorically in the sense of our previous research.

2. DEPENDENT TYPES

Dependent types are families of types indexed by terms [11]. They offer a degree of precision in describing program behaviours that goes far beyond the other typing features. In dependent type theory (DTT) a term variable $x : \sigma$ can occur in another type $\tau(x) : \texttt{Type}$. As an example we assume a type *IntList* of lists of integers with operations

$$\begin{array}{ll} \textit{nil} : & \textit{IntList} \\ \textit{append} : & \textit{Int}, \textit{IntList} \rightarrow \textit{IntList} \\ \textit{head} : & \textit{IntList} \rightarrow \textit{Int} \\ \textit{tail} : & \textit{IntList} \rightarrow \textit{IntList} \\ \textit{isempty} : & \textit{IntList} \rightarrow \textit{Bool} \end{array}$$

where *Int* and *Bool* are types of integers and boolean values, respectively. In DTT we can refine the type *IntList* to a family of types $\textit{IntList}(n)$, the types of lists with n elements, where $n : \textit{Nat}$ is a natural number. In such a manner we form a dependency of the type $\textit{IntList}(n)$ on the type $\textit{Nat}$. To express this dependency between arguments of operations and the types of their results, we consider e.g. that the type of operation *append* is a function $\textit{append} : \textit{Int}, \textit{IntList}(n) \rightarrow \textit{IntList}(\textit{succ}(n))$, where *succ* is an operation of type $\textit{Nat}$. It is clear that after appending an element to a list of n elements we get a list of $n + 1$ elements. So we capture in types the dependency between the value of an argument $n : \textit{Nat}$ on one side and the type $\textit{IntList}(n)$ and result type $\textit{IntList}(\textit{succ}(n))$ on the other side. Such types do not exist in ChTT and PTT. They can be considered to be similar as I-indexed collection $X = (X_i)_{i \in I}$ of sets, which can be written as

$$i : I \vdash X_i : \textit{Set}$$

where $\vdash I : Set$, i.e. I is an (index) set.

The types of operations for list of integers of length n can be written using a new constructor, *dependent product* $\prod$ as follows:

$$\begin{array}{ll} nil: & IntList(0) \\ append: & \prod n : Nat.(Int, IntList(n)) \to IntList(succ(n)) \\ head: & \prod n : Nat.IntList(succ(n)) \to Int \\ tail: & \prod n : Nat.IntList(succ(n)) \to IntList(n) \end{array}$$

The types of the operations $nil, append$ and *tail* tell us how many elements are in their results and that *head* and *tail* operations demand non-empty lists as arguments. Now we do not need the operation *isempty* because we can see whether the number of list n is 0. So dependent function types

$$\prod x : \sigma.\tau$$

are more precise form of function types $\sigma \to \tau$ of ChTT. The dependent product constructor binds a variable x representing the argument of the function so that we can mention it in the result type τ.

We can build also higher-level list manipulating operations with similarly refined types. For example, we can define a *new operation*, e.g. sorting function

$$sort : \prod n : Nat.IntList(n) \to IntList(n)$$

that returns a sorted list of the same length as the input. We can also construct *new terms*, e.g.

$$\begin{array}{ll} append3 = & \lambda n : Nat.\lambda i : Int.\lambda l.IntList(n). \\ & append(succ(succ(n))\ i \\ & (append(succ(n))\ i\ (append(succ(n))\ i\ (append(n\ i\ l)) \end{array}$$

which appends three integers in an integer list of type $IntList(n)$ of n elements and returns a list of $n + 3$ elements, i.e. of type $IntList(succ(succ(succ(n))))$.

Dependent types are widely used in computer science, e.g. in the description of digital systems we deal with types of bit vectors of a specific length $n : Nat$, i.e. types $BoolVec(n) = Bool^n$ that can be represented as n-tuples of boolean constants $true, false : Bool$ (or more conveniently $0, 1 : Bool$). The type $BoolVec(n)$ depends on $n : Nat$ [4]. Dependent type theory is often called Martin-Löf type theory [6] but his dependent type calculus contains also a type of all types that leads to Girard's paradox [2]. Dependent type theory is used not only for foundational reasoning [3, 5] but also as a basis for proof tools [10].

3. DEPENDENT TYPE CALCULUS

In this section we describe the syntax of dependent type calculus. In our considerations let Σ be a many-typed signature containing basic types. Because in dependent types can occur terms, types cannot be introduced separately, so recursion is required. Constructors for dependent types are:

- $\prod x : \sigma.\tau(x)$, i.e. *dependent product* of type $\tau(x)$, where term variable x ranges over type σ;
- $\sum x : \sigma.\tau(x)$, i.e. *dependent sum* of type $\tau(x)$, where term variable x ranges over type σ;
- $Eq_\sigma(x, x')$, i.e. the type of σ-equality for variables x, x' ranging over σ. Equality types are called *identity types.*

A dependent product is a collection of functions $(f)_\sigma$, i.e. indexed by σ, such that for every $i : \sigma$

$$f(i) : \tau[i/x]$$

is of type τ where occurrences of variable x are replaced by $i : \sigma$. A dependent sum is a set of pairs (i, j), where $i : \sigma$ and $j : [i/x]$. The substitution $[i/x]$ in type τ is typical for dependent type theory. Dependent products generalise exponents and dependent sums generalise Cartesian products of ChTT.

If x, x' are variables of the same type, the associated equality type is

$$Eq_\sigma(x, x') = \begin{cases} \{*\} & \text{if } x = x' \\ \emptyset & \text{otherwise} \end{cases}$$

where $\{*\}$is singleton.

We use *type context* $\Gamma = (x_1 : \sigma_1, \dots, x_n : \sigma)$ denoting a finite sequence of typed variables as in ChTT and PTT but we add the following property: every type σ_{i+1} is a well-formed type in the previous context $\Gamma' = (x_1 : \sigma_1, \dots, x_i : \sigma_i)$, i.e.

$$x_1 : \sigma_1, \dots, x_i : \sigma_i \ \vdash \ \sigma_{i+1} : \texttt{Type}$$

From this definition it follows that every free variable $y : \sigma_{i+1}$ must already have been declared in Γ', i.e. it must be one of $x_1, \dots, x_i$.

Example 1: The well-formed context may be e.g.

$$\Gamma = (n : Nat, l : IntList(n))$$

but

$$\Delta = (n : Nat, z : Array(n, m))$$

is not well-formed because m is not declared.

□

A *sequent of* DTT may have one of the following forms:

$$\begin{array}{lll} \Gamma \ \vdash \ \sigma : \texttt{Type} & & (1) \\ \Gamma \ \vdash \ t : \sigma & & (2) \\ \Gamma \ \vdash \ t = s : \sigma & & (3) \\ \Gamma \ \vdash \ \sigma = \tau : \texttt{Type} & & (4) \end{array}$$

The sequent (1) denotes dependent type σ in the context Γ, (2) denotes a term t of type σ in context Γ and (3) expresses equality (conversion) of terms. In (4) is described the conversion of types, because terms may occur in types. From categorical point of view types are equal if they are inhabited by the same terms.

Basic rules for dependent type theory are:

$$\frac{\Gamma \vdash \sigma : \mathtt{Type}}{\Gamma, x : \sigma \vdash x : \sigma}\ (proj) \qquad \frac{\Gamma \vdash t : \sigma \quad \Gamma, x : \sigma, \Delta \vdash \tau}{\Gamma, \Delta[\tau/x] \vdash \tau[t/x]}\ (subst)$$

$$\frac{\Gamma, x : \sigma, y : \sigma, \Delta \vdash \tau}{\Gamma, x : \sigma, \Delta[x/y] \vdash \tau[x/y]}\ (contr) \quad \frac{\Gamma \vdash \sigma : \mathtt{Type} \quad \Gamma \vdash \tau}{\Gamma, x : \sigma \vdash \tau}\ (weak)$$

$$\frac{\Gamma, x : \sigma, y : \tau, \Delta \vdash \tau}{\Gamma, y : \tau, x : \sigma, \Delta \vdash \tau}\ (exchange)$$

where τ is an arbitrary expression occurable on the right side of sequent and x is not free in τ. For singleton, i.e. *unit type* 1 we introduce the following rules

$$\frac{}{\vdash 1 : \mathtt{Type}} \qquad \frac{}{\vdash \langle\ \rangle : 1} \qquad \frac{\Gamma \vdash t : 1}{\Gamma \vdash t = \langle\ \rangle : 1}$$

We can form dependent product $\prod$, dependent coproduct $\sum$ and equality type by the following rules:

$$\frac{\Gamma, x : \sigma \vdash \tau : \mathtt{Type}}{\Gamma \vdash \prod x : \sigma.\tau : \mathtt{Type}} \qquad \frac{\Gamma, x : \sigma \vdash \tau : \mathtt{Type}}{\Gamma \vdash \sum x : \sigma.\tau : \mathtt{Type}}$$

$$\frac{\Gamma \vdash \sigma : \mathtt{Type}}{\Gamma, x : \sigma, x' : \sigma \vdash Eq_\sigma(x, x') : \mathtt{Type}}$$

These type constructors change the context. The variable $x : \sigma$ becomes bound in $\prod x : \sigma.\tau$ and $\sum x : \sigma.\tau$. Term substitution can be defined by

$$\begin{array}{ll} (\prod x : \sigma.\tau)[s/y] = & \prod x : \sigma[s/y].\tau[s/y] \\ (\sum x : \sigma.\tau)[s/y] = & \sum x : \sigma[s/y].\tau[s/y] \\ Eq_\sigma(x, x')[s/y] = & Eq_{\sigma[s/y]}(x[s/y], x'[s/y]) \end{array}$$

where in the first two lines we assume that y is different from x and x is not free in s.

Associated rules for terms are the following:

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma.t : \prod x : \sigma.\tau}\ (abstraction)$$

$$\frac{\Gamma \vdash t : \prod x : \sigma.\tau}{\Gamma \vdash t\ s : \tau[s/x]}\ (application)$$

$$\frac{\Gamma \vdash \sigma : \mathtt{Type} \quad \Gamma, x : \sigma \vdash \tau : \mathtt{Type}}{\Gamma, x : \sigma, y : \tau \vdash \langle x, y\rangle : \sum x : \sigma.\tau}\ (pairing)$$

$$\frac{\Gamma \ \vdash \ \rho : \texttt{Type} \ \Gamma, x : \sigma, y : \tau \ \vdash \ s : \rho}{\Gamma, z : \sum x : \sigma.\tau \ \vdash \ (\texttt{unp}\ z \ \texttt{as} \ \langle x, y\rangle \ in \ s) : \rho} \ (unpairing)$$

$$\frac{\Gamma \ \vdash \ \sigma : \texttt{Type}}{\Gamma, x : \sigma \ \vdash \ \texttt{refl}_\sigma(x) : Eq_\sigma(x, x)} \ (reflexivity)$$

$$\frac{\Gamma, x : \sigma, x' : \sigma, \Delta \ \vdash \ \rho : \texttt{Type} \qquad \Gamma, x : \sigma, \Delta[x/x'] \ \vdash \ s : \rho[x/x']}{\Gamma, x : \sigma, x' : \sigma, z : Eq_\sigma(x, x'), \Delta \ \vdash \ (s \ \texttt{with} \ x = x' \ \texttt{via} \ z) : \rho}$$

where unp is unpairing operator for sum types similar as in [7], $\texttt{refl}_\sigma$ is reflexivity combinator for equality and s with $x' = x$ via z denotes the elimination term for dependent equality types.

4. ENCLOSING DEPENDENT TYPES IN CATEGORY

We start the categorical investigation of type dependency. We use a distinguished class of morphisms, *display maps* [14] in a category of contexts. A display map $\varphi : (X_i)_{i\in I} \to I$ in set theoretical sense is a mapping from a family $(X_i)_{i\in I}$ of sets to the set I. Then every set $X_i = \varphi^{-1}(i)$ is indexed by the element $i \in I$. Such indexing is equivalent with obvious pointwise indexing but it has a big advantage if considering categories of contexts. Every indexed set X_i can be regarded as a *fibre* subcategory over an object i and φ displays the (total) category $(X_i)_{i\in I}$ over a base category I.

Before we construct DTT categorically, we must consider the following fact: since terms can occur in types we may have conversions between types. But then it is possible to have conversions between contexts (componentwise). Therefore we do not consider contexts Γ as objects of classifying category, but equivalence classes $[\Gamma]$ of contexts w.r.t. conversion. But for notation simplicity we use in the following Γ for $[\Gamma]$.

We assume a fixed dependent type calculus over a signature Σ and we form the classifying category of contexts $ClD(\Sigma)$ for DTT that contains:

- as category objects equivalence classes $\Gamma, \Delta, \ldots$;
- as category morphisms $\Gamma \to \Delta$ (where $\Delta = (y_1 : \tau_1, \ldots, y_m : \tau_m)$, y_i are free in
τ_i) n-tuples $(t_1, \ldots, t_n)$ of terms t_i such that

$$\Gamma \ \vdash \ t_i : \tau_i[t_1/y_1, \ldots, t_{i-1}/y_{i-1}]$$

We denote this n-tuple of terms by $\vec{t} : \Gamma \to \Delta$ and call it a *context morphism.* Explicit substitution in types is typical for DTT and is not in ChTT and PTT. These substitutions are performed simultaneously.

Identity $\Gamma \to \Gamma$ for every context $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ is the n-tuple $(x_1, \ldots, x_n)$ of variables.

Composition of morphisms

$$\Gamma \xrightarrow{(t_1,\ldots,t_m)} \Delta \xrightarrow{(s_1,\ldots,s_k)} \Theta$$

where

$$\begin{aligned} \Gamma = &\quad (x_1 : \sigma_1, \ldots, x_n : \sigma_n) \\ \Delta = &\quad (y_1 : \tau_1, \ldots, y_m : \tau_m) \\ \Theta = &\quad (z_1 : \rho_1, \ldots, z_k : \rho_k) \end{aligned}$$

and

$$\begin{aligned} &\Gamma \vdash t_i : \tau_i[t_1/y_1, \ldots, t_{i-1}/y_{i-1}] \\ &\Delta \vdash s_j : \rho_j[s_1/z_1, \ldots, s_{j-1}/z_{j-1}] \end{aligned}$$

is the k-tuple $(u_1, \ldots, u_k) : \Gamma \to \Theta$ with components

$$u_j = s_j[\vec{t}/\vec{y}] = s_j[t_1/y_1, \ldots, t_m/y_m]$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, k$, that are well-typed, i.e.

$$\begin{aligned} \Gamma \ \vdash \quad & u_j : \rho_j[s_1/z_1, \ldots, s_{j-1}/z_{j-1}]\ [\vec{t}/\vec{y}] \\ & = \rho_j[s_1/z_1, \ldots, s_{j-1}/z_{j-1}] \end{aligned}$$

To prove associativity is not so simple and the method of the proof can be found in [14, 12]. Now we can say that $ClD(\Sigma)$ constructed above is a category.

In ChTT and PTT classifying categories have finite products, empty type context is terminal object and concatenation of contexts yields binary products. In DTT is easy to see that empty context again yields a terminal object. But concatenation of contexts does not yield products, but rather dependent sums.

Example 2: Let $(x : \sigma, y : \tau)$ be a type context of two types, where x may occur in τ. A context morphism $\Gamma \to (x : \sigma, y : \tau)$ does not correspond to two morphisms

$$\Gamma \to (x : \sigma) \qquad \Gamma \to (y : \tau)$$

but to the following two morphisms

$$t : \Gamma \to (x : \sigma) \qquad s : \Gamma \to (y : \tau[t/x]) \qquad (*)$$

This dependent pairing property can be described by the existence of pullbacks in the category $ClD(\Sigma)$ along display maps:

$$(\Gamma, z : \rho) \xrightarrow{\varphi} \Gamma.$$

Explicitly, for $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ the map $\varphi : (\Gamma, z : \rho) \to \Gamma$ is the n-tuple $(x_1, \ldots, x_n)$ of variables in Γ. Display map is some kind of (dependent) projection because all variables declared in Γ may occur free in ρ. This situation is illustrated in Figure 1, where display maps are closed under pullback (t, s), where $(t, s) : \Gamma \to (x : \sigma, y : \tau)$ is a context morphism and t and s are as in $(*)$.

$\square$

$$\begin{tikzcd}
\Gamma \arrow[dr, dashed, "(\vec{x},s)"] \arrow[drr, "(t,s)"] \arrow[ddr, "id"'] & & \\
& \Gamma, y:\tau[t/x] \arrow[r] \arrow[d, "\varphi"'] \arrow[dr, phantom, "\lrcorner", very near start] & (x:\sigma, y:\tau) \arrow[d, "\varphi"] \\
& \Gamma \arrow[r] & (x:\sigma)
\end{tikzcd}$$

Figure 1: Display map closed under pullback

We denote by $\mathcal{D}$ the collection of display maps

$$\varphi : (\Gamma, x : \sigma) \to \Gamma$$

in $ClD(\Sigma)$ induced by types $\Gamma \vdash \sigma :$ `Type` in contexts. We construct the *arrow category* $\mathcal{D}^{\to}$ consisting of

- display maps φ as objects and
- pairs of morphisms $(u, g) : \varphi \to \psi$ as category morphisms, where u, g are as in the commutative diagram in Figure 2.

$$\begin{tikzcd}
\Gamma, x:\sigma \arrow[r, "g"] \arrow[d, "\varphi"'] & \Delta, y:\tau \arrow[d, "\psi"'] \\
\Gamma \arrow[r, "u"'] & \Delta
\end{tikzcd}$$

Figure 2: Morphisms in $\mathcal{D}^{\to}$

Maps in $\mathcal{D}$ form a split fibration over $ClD(\Sigma)$ in Figure 3.

$$\begin{tikzcd}
\mathcal{D}^{\to} \arrow[d, "p"'] \\
ClD(\Sigma)
\end{tikzcd}$$

Figure 3: Dependent types fibration

where p assigns to every display map $\varphi : \Gamma, x : \sigma \to \Gamma$ the codomain context, $p(\varphi) = \Gamma$. Substitution functor $\varphi^* : \mathcal{D}^{\to}_{\Gamma} \to \mathcal{D}^{\to}_{\Gamma,x:\sigma}$ along a display map $\varphi : (\Gamma, x : \sigma) \to \Gamma$ in this fibration is functor between corresponding fibre subcategories over corresponding contexts. It is *weakening* because it moves a type $\Gamma \vdash \tau :$ `Type` to a bigger context $\Gamma, x : \sigma \vdash \tau :$ `Type` as in the pullback in Figure 4.

$$\begin{tikzcd}
(\Gamma, x:\sigma, y:\tau) \arrow[r] \arrow[d, "\varphi^*(\psi)"'] & (\Gamma, y:\tau) \arrow[d, "\psi"] \\
(\Gamma, x:\sigma) \arrow[r, "\varphi"'] & \Gamma
\end{tikzcd}$$

Figure 4: Substitution functor φ^*

A unit type $1:\tau$ corresponds to a terminal object functor $\mathbf{1} : ClD(\Sigma) \to \mathcal{D}^{\to}$ for the fibration of display maps. This functor is defined by

$$\Gamma \mapsto \begin{pmatrix} \Gamma, z:1 \\ \downarrow \\ \Gamma \end{pmatrix}$$

Then for arbitrary display map $\varphi : (\Gamma, x:\sigma) \to \Gamma$ there is precisely one pair of morphisms

$$\begin{tikzcd}
(\Gamma, x:\sigma) \arrow[d] \arrow[r, "{(x_1,\ldots,x_n,\langle\,\rangle)}"'] & (\Gamma, z:1) \arrow[d] \\
\Gamma & \Gamma
\end{tikzcd}$$

where $x_1, \ldots, x_n$ are variables declared in Γ.

Dependent products $\prod$ and sums $\sum$ correspond to the fibration in Figure 2 having Cartesian products and sums along display maps $\varphi : (\Gamma, x:\sigma) \to \Gamma$ in $ClD(\Sigma)$. This means that dependent products correspond to right adjoints of weakening functors φ^* along display maps and dependent sums $\sum$ correspond to left adjoints along display maps. For dependent products and sums must hold Beck-Chevalley conditions [4], i.e. for any morphism $t : \Gamma, x:\sigma \to \Gamma$ in $ClD(\Sigma)$ and every pair of reindexing (substitution) functors $\varphi^*, \varphi^{\#} : \mathcal{D}^{\to}_{\Gamma} \to \mathcal{D}^{\to}_{\Gamma,x:\sigma}$ between fibres, natural transformation is an identity.

We can say that the codomain fibration in Figure 2 from display maps arrow category to classifying category characterizes categorically DTT. Every object Γ in classifying category indexes a fibre subcategory $\mathcal{D}^{\to}_{\Gamma}$. Dependent type constructors $1, \prod$ and $\sum$ are defined by adjoints to substitution functors in total category $\mathcal{D}^{\to}$.

5. Conclusion

In this contribution we presented dependent type theory categorically as fibration from arrow category of display maps to classifying category consisting of dependent type contexts. We can say now that we have integrated categorical approach for representing ChTT, PTT and DTT. Over these type theories we construct logical system also as fibration similarly as in [9], i.e. in the case of DTT it will be double fibration over classifying category. In the following research we would like to extend this approach also to higher-order dependent type theory, based on polymorphic and dependent types.

This work is supported by the grant VEGA 1/2181/05: Mathematical theory of programming and its application in the methods of stochastic programming

References

[1] Th.Ehrhard: A categorical semantics of constructions, In: Logic in Computer Science, IEEE, Computer Science Press, 1988, pp.264-273

[2] J.-Y.Girard: Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'order supéieur, PhD.Thesis, Université Paris, 1972

[3] M.Hofmann: Syntax and semantics of dependent types, Semantics and Logic of Computation, Cambridge Univ.Press, 1997

[4] B.Jacobs: Categorical logic and type theory, Elsevier, Amsterdam, 1999

[5] Z.Luo: Computation and reasoning: A type theory for computer science, Monographs on Computer Science, Oxford Univ.Press, 1994

[6] P.Martin-Löf: Intuitionistic type theory, Bibliopolis, Napoli, 1984

[7] V.Novitzká: Church's types in logical reasoning on programming, Acta Electronica et Informatica, Vol.6,No.2,2006, Košice, pp.27-31

[8] V.Novitzká, D.Mihályi, V.Slodičák: Categorical logic over Church's types, Proc. 6th Scient.Conf. Electronic Computers and Informatics ECI'2006, Košice-Herľany, September 2006, pp.122-129

[9] V.Novitzká, D.Mihályi: Polymorphic type theory for categorical logic, Acta Electrotechnica et Informatica, Kosice, To appear, 2007

[10] S.Owre et al: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS, IEEE Trans. on Softw.Eng., Vol.21, No.2, 1995, pp.107-125

[11] B.C.Pierce: Types and programming languages, MIT, 2002

[12] A.M.Pitts: Categorical logic, In: S.Abramsky, D.M.Gabbai, T.S.E.Maibaum (eds.): Handbook of Logic in Computer Science, Vol.6, Oxford Univ.Press, 1995

[13] P.Taylor: Recursive domains, indexed category theory and polymorphism, PhD.Thesis, Univ.Cambridge, 1986

[14] P.Taylor: Practical foundations of mathematics, Cambridge Univ.Press, 1999

[15] Th.Streicher: Semantics of type theory. Correctness, completeness and independence results, Progress in Theoretical ComputerScience, Birkhauser, Boston, 1991

Faculty of Electrical Engineering and Informatics, Technical University of Košice
E-mail address: `valerie.novitzka@tuke.sk, anita.verbova@tuke.sk`