5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics
January 25-26, 2007 ░ Poprad, Slovakia

Categorical Type Analysis for Parsing Algebras

Daniel Mihályi, Valerie Novitzká, Viliam Slodičák

Department of Computers and Informatics
Technical University Košice
Letná 9, 042 10 Košice, Slovakia

Abstract: Solving the problems by computers we always start with formulation of their theoretical foundations. We use necessary mathematical theories and reason in them to obtain expected results. In this contribution we assume a theoretical description of programming process as logical reasoning over axiomatized theories. We consider intuitionistic logical reasoning over type theory of solved problem. We formulate type theory in categorical terms and construct its model as a functor. We use type automata for constructing a mechanism for type checking and formulate a relation with type theory as a special mapping that generate to every well-formed typed term of our type theory a typed automaton recognizing it.

Keywords: type theory, category theory, typed automata

1 Introduction

The aim of our research is the theoretical description of problem solving process in which we would like to use computers. Computers execute programs that can be shortly characterized as data structures and algorithms. Data structures have always some types therefore we can formulate a type theory for a given solved problem. Because types can be complex structures, we use for them cartesian closed category. We consider computations in programs as logical reasoning in an intuitionistic logical system over corresponding type theory in the framework of some axiomatized mathematical theories. Curry-Howard correspondence [3] can help us in our considerations, because types correspond to propositions and computations in term calculus correspond to mappings between propositions, i.e. proofs.

In this contribution we formulate a type theory over a many-typed signature of a given solved problem and we enclose it into a classifying category. Then we define a model of type theory as a functor into cartesian closed category contaning carrier sets, i.e. type representations as objects. There are many approaches that

use another representation for type theory, e.g. in [8,9] but they are more pragmatical than our theoretical approach.

Typed terms can be infered by typed automata and evaluation of terms can be described by tree automata. In this contribution we consider typed automata introduced in [4,5] used for recognition languages with structure. Our approach differs from the origin aim of typed automata because of their availaibility for analysing well-formedness and type-checking of terms of our type theory. Typed automata are finite deterministic automata with the additional information. In our approach this additional information is type and symbols of input alphabet (variables and function symbols) bring their types into a typed automaton recognizing given term. We can say that a term is well-formed if the appropriate automata terminates its work and a final state is of the expected type. States in typed automata are typed and serve for type checking of terms. Typed automata can be depicted in the category ***Set*** of sets as objects and functions as morphisms. This category is cartesian closed category and we construct a functor from classifying category to ***Set*** expressing the relation between type theory and typed automata. The existence of such functor gives us the possibility using typed automata for type checking in later phases of program construction. Moreover, typing states of deterministic finite automata has crucial influence on speed of string processing and significant reduction of search space [4].

2 Type Theory and Typed Algebraic Structures

In our approach we enclose the names of types and operations on them needed for solving problem into a well-known notion of algebraic specification, *many-typed signature.* A many-typed signature $\Sigma = (T, F)$ consists of a finite set $T = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ of *basic types* and a finite set F of *function symbols* of the following form

$$f\colon \sigma_1 \ldots \sigma_n \to \sigma_{n+1}.$$

A *signature morphism* $\phi : \Sigma_1 \to \Sigma_2$ from a signature $\Sigma_1 = (T_1, F_1)$ to a signature $\Sigma_2 = (T_2, F_2)$ is a pair $(u, (f_\alpha))$, where $u{:}T_1 \to T_2$ is a function between sets of types and (f_α) is a family of functions between corresponding function symbols where $\alpha = ((\sigma_1, \ldots, \sigma_n), \sigma_{n+1})$. For instance, if $f\colon \sigma_1 \ldots \sigma_n \to \sigma_{n+1}$ then

$$f_\alpha(f) : u(\sigma_1), \ldots, u(\sigma_n) \to u(\sigma_{n+1}).$$

A signature Σ together with a set Ax of axioms formulated in some logical system and describing properties of operations form an *algebraic specification* (Σ, Ax). All signatures can be enclosed into the category ***Sign*** of signatures as objects and signature morphisms as category morphisms.

From basic types we can form more complex *Church's types* using constructors '$\times$' for *product types* $\sigma \to \tau$, '+' for *coproduct types* $\sigma + \tau$ and '$\to$' for *function*

types $\sigma\to\tau$ [6,7]. To introduce terms we need a countable set $Var = \{v_1, v_2, ..., w_1, w_2, ..., x, y, z, ...\}$ of *term variables*. Every variable has assigned exactly one (possibly Church's) type constructed from types of the set *T*. We denote by $\Gamma = (v_1 : \sigma_1, ..., v_n : \sigma_n)$ a sequence of *variable declarations* and call it *type context.* A *term* t of a type τ in which can occur only variables from the type context Γ is a sequent

$$\Gamma \vdash t : \tau .$$

Rules for constructing well-formed typed terms together with axioms of type theory are in [6]. We enclose all type contexts over a signature Σ into a *classifying category* $Cl(\Sigma)$ that contains

- type contexts $\Gamma = (v_1 : \sigma_1, ..., v_n : \sigma_n)$, $\Delta = (w_1 : \tau_1, ..., w_n : \tau_m)$ as category objects;
- terms $t: \Gamma \to \Delta$ as category morphisms, where $\Gamma \vdash t : \tau$ and $\tau \in \Delta$.

We investigate the classifying category $Cl(\Sigma)$. Every its object, a type context $\Gamma = (v_1 : \sigma_1, ..., v_n : \sigma_n)$ can be considered as a single variable declaration $v : \sigma_1 \times ... \times \sigma_n$ of product type. Function types ensure the existence of exponent objects Γ^{Δ} in $Cl(\Sigma)$. The empty product denoted by 1 is a terminal object of classifying category, i.e. there exists exactly one morphism from any object of $Cl(\Sigma)$ to the object 1. Therefore we can consider that classifying category $Cl(\Sigma)$ for any many-typed signature Σ is cartesian closed category (ccc).

A model of an algebraic specification (Σ, Ax) is a Σ- algebra. To every type $\sigma \in T$ we assign a *carrier set* A_σ and to every function symbol $f: \sigma_1 ... \sigma_n \to \sigma_{n+1}$ we assign a function (*operation*)

$$f^A: A_{\sigma 1} \times ... \times A_{\sigma n} \to A_{\sigma n+1}$$

between corresponding carrier sets. Σ- model is a Σ- algebra

$$((A_\sigma)_{\sigma \in T}, (f^A)_{f \in F})$$

whose operations satisfy axioms from *Ax*.

Considering models of all many-typed signatures we can construct set-based category of algebraic models ***AlgMod*** as follows

- objects are 3-tuples $(\Sigma, (A_\sigma)_{\sigma \in T}, (f^A)_{f \in F})$, where $((A_\sigma)_{\sigma \in T}, (f^A)_{f \in F})$ is a model of a signature Σ;
- morphisms

$$(\phi, (h_\sigma)_{\sigma \in T}): (\Sigma, (A_\sigma)_{\sigma \in T}, (f^A)_{f \in F}) \to (\Sigma', (A'_\sigma)_{\sigma \in T}, (f'^A)_{f \in F})$$

are pairs consisting of a signature morphism $\phi: \Sigma \to \Sigma'$ and a family of functions $h_\sigma : A_\sigma \to A'_{\phi(\sigma)}$ such that the diagram in Figure 1 commutes.

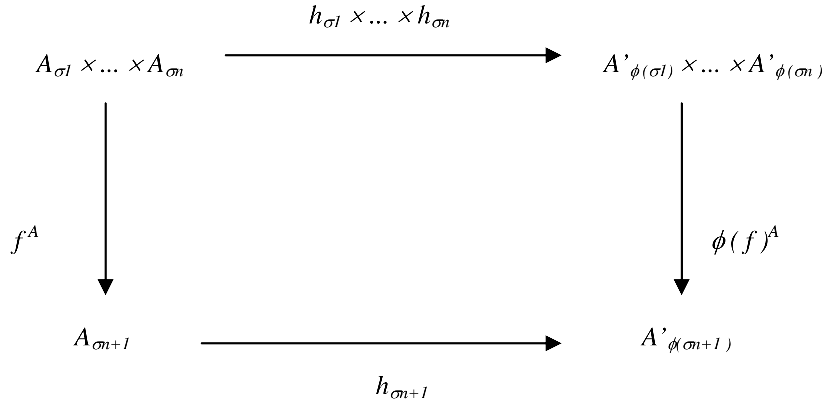

Figure 1
Morphisms in the category ***AlgMod***

This algebraic approach can be extended to more complex structures using category theory. Let $\boldsymbol{B}$ be a cartesian closed category. We construct a functor $M: Cl(\Sigma) \rightarrow \boldsymbol{B}$ as follows:

- to every object in $Cl(\Sigma)$, a type context $\Gamma = (v_1 : \sigma_1, \ldots, v_n : \sigma)$ we assign a carrierset product

  $A_{\sigma 1} \times \ldots \times A_{\sigma n}$ in $\boldsymbol{B}$;

- to every morphism $\Gamma \rightarrow \Delta$ in $Cl(\Sigma)$, a term $\Gamma \mid\!\!- t: \tau,\ t \in \Delta$ we assign a function

$$\lambda (a_1, \ldots, a_n) . M(t[a_1/ v_1, \ldots, a_n/ v_n])$$

where $a_i \in A_{\sigma i}$, for $i = 1, \ldots, n$ are elements of corresponding carriersets, i.e. values substituted to variables of appropriate types.

A model of a type theory described by a classifying category $Cl(\Sigma)$ is a functor $M: Cl(\Sigma) \rightarrow \boldsymbol{B}$ defined above into a cartesian closed category that preserves products [6]. This type theory and its semantics we used in [7] to construct logical system and its semantics above type theory as a base category in fibrations.

3 Typed Automata

A typed automaton is a deterministic finite automaton with types. Input symbols not only follow each other but have a type and so force the automaton to respect some rules. Typed automaton is able to recognize not only a string of symbols from input alphabet but to analyze its internal structure. For these purposes input data bring some additional information that help to recognize internal structure of

input string and this information is appended to states. Typed automata were defined in [4] for grammatical inference in learning formal grammars for unknown languages from given learning data.

In our approach we assume that the additional information appended to input symbols is the knowledge that input symbols (terms) are typed. As input alphabet we use typed variables, function symbols from many-typed signature and brackets as auxiliary symbols. Opening brackets are dummy symbols for typed automata, closing brackets inform about finishing analysis of a function symbol. Typed automaton checks whether the input string is well-formed term, i.e. it is constructed by term construction rules and is well-typed. For example, the input string of the form *plus)x* is not well-formed term and we cannot find typed automaton accepting it. But the input string *plus(b,x)*, where *b:boolean* is a variable of type *boolean* and *x:integer* is a variable of type *integer* is also not well-formed and is not accepted by any typed automaton.

We use typed automata to analyze a construction of well-formed terms of type system introduced in the previous section. Let $\Sigma = (T, F)$ be a many-typed signature, where $T = \{\sigma_1, ..., \sigma_n\}$ is a finite set of types and F is a family of function symbols of the form $f: \sigma_1 \ldots \sigma_m \to \sigma_{m+1}$. We define the set F' containing only names of function symbols from signature Σ and we get as the input alphabet of our typed automaton the set

$$\Sigma' = \Sigma \cup Var \cup F'$$

where Var is the set of typed variables.

Definition 1 A *typed automaton TA* is a 7-tuple

$$TA = (\Sigma', Q, q_0, Q_{final}, \delta, T', type)$$

where

Σ'	is an input alphabet
Q	is a finite set of states
q_0	is the initial state
$\delta: Q \times \Sigma' \to Q$	is a transition (next state) function
$T' = T \cup \{\sigma_{init}\}$	is a set of types from Σ extended with the special type σ_{init}
$type: Q \to T'$	is a typing function assigning to every state just one type from T'
$Q_{final} \subseteq Q$	is a set of final states

A language $L(TA)$ defined by the typed automaton TA is a set of all words w in $(\Sigma')^*$ such that

$$\delta(q_0, w) = q,$$

where $q \in Q_{final}$ is a final state. We can extend the definition of typing function to words as follows. For all subwords $w \in Prefix(L(TA))$ of words recognizable by typed automata TA

$$type(w) = type(\delta(q_0, w)).$$

Example 1 Let $\Sigma = (T, F)$ be a many-typed signature, where $T = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ are types and $F = \{f: \sigma_1\ \sigma_4 \rightarrow \sigma_5,\ g: \sigma_2\ \sigma_3 \rightarrow \sigma_4\}$ are function symbols. Let

$$x: \sigma_1, y: \sigma_2, z: \sigma_3 \vdash f(x, g(y,z)) : \sigma_5$$

be a term of type σ_5 . A posible typed automaton TA analyzing well-formedness of this term can be constructed as follows.

We define

- the input alphabet $\Sigma' = \{x, y, z\} \cup \{f, g\} \cup \{(,)\}$
- the set of states $Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$
- the set of final states $Q_{final} = \{q_8\}$
- the initial state q_0
- the set of types $T' = T \cup \{\sigma_{init}\}$
- transition function $\delta : Q \times \Sigma \rightarrow Q$ by the equations

$\delta(q_1, f(\,) = q_2$ $\delta(q_2, x) = q_3$ $\delta(q_3, g(\,) = q_4$ $\delta(q_4, y) = q_5$

$\delta(q_5, z) = q_6$ $\delta(q_6,)) = q_7$ $\delta(q_7,)) = q_8$

- and the typing function $type : Q \rightarrow T'$ by the equations

$type(q_1) = \sigma_{init}$ $type(q_2) = \sigma_1 \times \sigma_4 \rightarrow \sigma_5$

$type(q_3) = \sigma_1$ $type(q_4) = \sigma_2 \times \sigma_3 \rightarrow \sigma_4$

$type(q_5) = \sigma_2$ $type(q_6) = \sigma_3$

$type(q_7) = \sigma_4$ $type(q_8) = \sigma_5$

The typed automaton $TA = (\Sigma', Q, q_0, Q_{final}, \delta, T', type)$ constructed above is illustrated in Figure 1.

We note that the typed automaton in Figure 2 can be used also to analyze a term $x{:}\sigma_1, x'{:}\ \sigma_4 \vdash f(x, x') : \sigma_5$ and the input symbol $x'{:}\ \sigma_4$ causes that $\delta(q_2, x') = q_7$ and $type(q_7) = \sigma_4$.

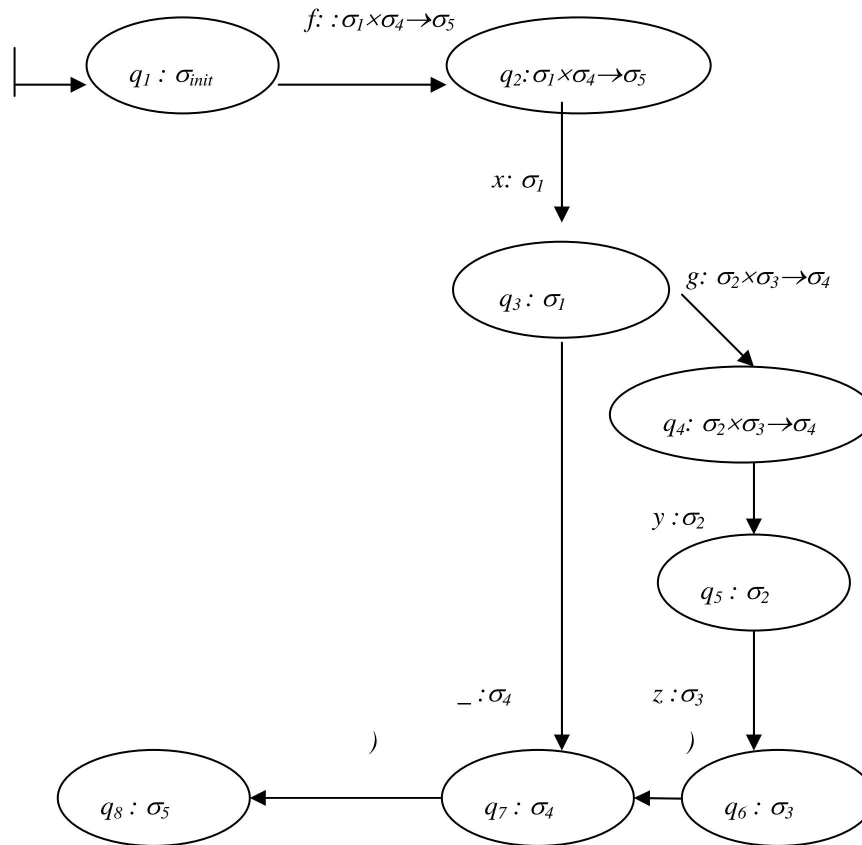

Figure 2
Typed automaton for well-formed term

4 From Type Theory to Typed Automata

Every automaton can be depicted in the category ***Set*** consisting of sets as category objects and functions between sets as category morphisms [1]. The initial object of the category ***Set*** is the empty set and terminal objects are singleton sets denoted by *{*}*. The category ***Set*** is cartesian closed category, i.e. it has products and exponential objects [2]. A typed automaton $TA = (\Sigma', Q , q_0 , Q_{final} , \delta, T', type)$ can be depicted in ***Set*** as it is shown in Figure 3.

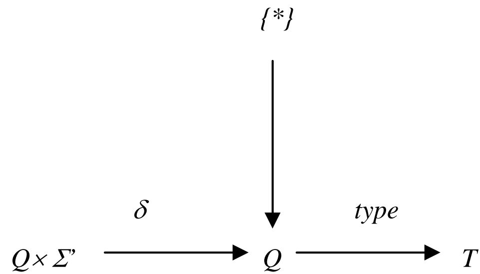

Figure 3
Typed automaton in the category ***Set***

Q, T' and *{*}* are sets, i.e. they are the objects in ***Set,*** $Q\times Q$ is a product object and δ*, type* are functions between sets, i.e. morphisms in the category ***Set.*** A terminal object *{*}* denotes a domain for initializing function to the initial state q_0 of *TA*.

Now we define how to find for every well-formed term $t:\tau$, i.e. a category morphism in $Cl\ (\Sigma\ )$ a corresponding typed automaton. In the category ***Set*** there always exists a set $T' = T \cup \{\sigma_{init}\}$, where *T* contains all types in Γ and the type $\tau \in \Delta$. Similarly, because ***Set*** has as objects *all* sets, there exists a set $\Sigma' = \Sigma \cup Var \cup F'$ containing all symbols occuring in *t*.

We get as a set *Q* of states the least set such that every its element has a type from *T'*. Then we have also a the type function $type: Q \rightarrow T'$ assigning to every state its type. The initialization function $\{\ *\ \} \rightarrow Q$ assigns to terminal object, singleton set, the initial state from *Q*. A morphism from terminal object to any object of category provides a constant and in ***Set*** always exists such morphism.

Because ***Set*** is cartesian closed category, to any pair of objects there exists an object, their product. From this fact it follows that $Q \times \Sigma'$ is also an object in ***Set***. We define a function $\delta: Q \times \Sigma'$ that has to satisfy the following property:

$$\delta(\ q_0\ ,\ t\ ) = q \quad \text{and} \quad type\ (q) = \tau\ .$$

Then we get as Q_{final} a singleton set containig only the state *q* of type τ.

In such a manner we can always find in the category ***Set*** a typed automaton reecognizing any well-formed term of our type theory and we have got a useful mechanism for type analysis of terms.

Conclusion

In our contribution we formulated a type theory over solved problem in the framework of category theory. Such type theory represented by classifying category form a base category in fibration, a special functor that enable to formulate typed logical system over given type theory. We defined a model of

type theory as the category ***AlgMod*** of algebras. Then we investigated the properties of finite deterministic automata and show that typed automata can help us to analyse well-formedness of typed terms and make type checking. We showed that to every term in type theory that is a morphism in the classifying category we can find a type automaton recognizing this term.

We would like to follow our research with the equipment tree automata with types and use them for describing evaluation of terms.

Acknowledgment

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] Adámek J., Trnková V.: Automata and Algebras in Categories, Kluver Academic Publishers, Doodrecht, 1989

[2] Barr, M., Wells, Ch.: Category Theory for Computing Science. Prentice-Hall, 1990

[3] Girard, J.-Y., Taylor P., Lafont Y.: Proofs and Types, Cambridge Univ. Press, 1990

[4] Kermorvant Ch., Higuera C.: Learning Languages with Help, In: Proc. International Coll. On Grammatical Inference, Vol. 2484 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2002, pp. 161-173

[5] Kermorvant Ch., Higuera C., Dupont P.: Learning Typed Automata from Automatically Labeled Data, Journal Électronique d'Intelligence Artificielle, Vol. 6, 45, 2004

[6] Novitzká Valerie: Church's Types in Logical Reasoning on Programming, Acta Electrotechnica et Informatica, Vol. 6, No. 2, 2006, Košice, pp. 27-31

[7] Novitzká Valerie, Mihályi Daniel, Slodičák Viliam: Categorical Logic over Church's Types, Proc. 6th International Scient. Conference Electronic Computers and Informatics ECI'2006, Košice-Herľany, September 20-22, 2006, Košice, 2006, pp. 122-127

[8] Porkoláb, Z., Zólyomi, I.: A Feature Composition Problem and a Solution Based on C++ Template Metaprogramming, Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, Vol. 4143, Springer-Verlag, 2006, pp. 459-470

[9] Zólyomi, I., Porkoláb, Z., Kozsik, T.: An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming, Generative Programming and Component Engineering, Lecture Notes in Computer Science, Vol. 2830, Springer-Verlag, 2003, pp. 209-227