Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 1. pp. 91–98.

Finite automata in the mathematical theory of programming[*]

Valerie Novitzká, Daniel Mihályi, Viliam Slodičák

Technical University of Košice
e-mail: {valerie.novitzka,daniel.mihalyi,viliam.slodicak}@tuke.sk

Abstract

Finite deterministic sequential and tree automata are well-known structures that provide mechanisms to recognize languages. We extend these automata to typed ones over our type theory of solved problem enclosed in classifying category. The main idea is that symbols of input alphabet bring some additional information, types of symbols, and this information is appended to the states of automata. Typed sequential automata can serve for analyzing well-formedness of terms and typed tree automata can be used for evaluation of typed terms of our type theory. We show how typed automata can be depicted in the category **Set** of sets and what is the relation between our type theory and typed automata in categorical terms.

Keywords: automata, category theory

MSC: 03D40, 18A15

1. Introduction

The aim of our research is theoretical description of program development process. We consider programming as logical reasoning in some intuitionistic logical system over a type theory for a given solved problem and we formulate them as fibrations in category theory [7]. In this paper we present a part of our results. We use Church's type theory defined in [6] that is enclosed in a classifying category with variable declarations as objects and typed terms as morphisms. Another approach to type abstractions can be found in [8] and more practical handling with types is in [9].

Another useful mechanism for handling terms provide finite automata. In our approach the variables, function symbols and terms have types, therefore we need to introduce types into automata. We follow the way sketched in [4, 3]. The main

[*]This work was supported by VEGA Grant No.1/2181/05: Mathematical theory of programming and its application in the methods of stochastic programming.

idea of the authors is that the input symbols bring some additional information enabling the automata to recognize internal structure of input strings. That means, the input symbols not only follow each other but get an information of the structure of substrings and so force the automaton to respect some rules. Such typed automata were defined for recognizing languages with structure. For our purposes the additional information brought by input symbols are types and they are appended to every state of typed automaton.

Typed sequential automata can serve for checking well-formedness of typed terms and typed tree automata can be used for their evaluation. Typed automata can be depicted in the category **Set** of sets and we define a mapping that assigns to every typed term in classifying category a typed finite sequential/tree automaton accepting it in the category **Set**.

2. Type theory and its model

Church's type theory can be enclosed into classifying category $Cl(\Sigma)$ over a signature $\Sigma = (T, F)$, where T contains basic types $\sigma, \tau, \ldots$ of a given problem and F is a finite set of operations of the form $f : \sigma_1 \ldots \sigma_n \to \tau$. We associate with a signature Σ two functions:

• *arity* : $F \to T^*$, where T^* is a set of finite strings over T. This function assigns to every function symbol $f : \sigma_1 \ldots \sigma_n \to \tau$ in F its arity, i.e. $arity(f) = \sigma_1 \ldots \sigma_n$;

• *type* : $F \to T$, that assigns to every operation specification its result type, i.e. $type(f) = \tau$.

From basic types we construct Church's types using constructors '$\times$' for product types, '$+$' for coproduct types and '$\to$' for function types and we denote by $\bar{T}$ the least set of Church's types over Σ. We assume a countable set $Var = \{v_1 : \sigma_1, v_2 : \sigma_2, \ldots, x, y, z, \ldots\}$ of *term variables.* Every variable has assigned exactly one type and the notation $v : \sigma$ declares the variable v of the type σ. A finite sequence

$$\Gamma = (v_1 : \sigma_1, \ldots, v_n : \sigma_n)$$

consisting of variable declarations is *type context.* A *term* t of type τ in which can occur only variables declared in Γ we denote by the following sequent

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

The *classifying category* $Cl(\Sigma)$ over a signature Σ has type contexts $\Gamma, \Delta, \ldots$ as category objects and typed terms $[t] : \Gamma \to \Delta$ as category morphisms, where $\Gamma \vdash t : \tau$ and $\tau \in \Delta$. The empty product type 1 is the terminal object and the empty coproduct type 0 is the initial object of the category $Cl(\Sigma)$. From the construction we see that $Cl(\Sigma)$ has finite products, finite coproducts, exponent objects $\sigma \to \tau$, terminal and initial object, therefore it is *bicartesian closed category* (biccc) [2].

A *model* of type theory over Σ is a functor $\mathcal{M}$ from the classifying category $Cl(\Sigma)$ to the biccc $\mathbf{B}$ of type representations

$$\mathcal{M} \colon Cl(\Sigma) \to \mathbf{B}$$

defined as follows:

- for every basic type σ in T, $\mathcal{M}(\sigma) = [\![\sigma]\!]$, an object in $\mathbf{B}$, a type representation of σ;
- for every operation specification $f\colon \sigma_1 \dots \sigma_n \to \tau$

$$\mathcal{M}(f) = [\![f]\!]\colon [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \to [\![\tau]\!],$$

a morphism between corresponding type representations.

3. Introducing types into sequential automata

Terms can be considered as strings over some input alphabet that have some internal structure. For recognizing strings of symbols over some alphabet we can use finite sequential automata [1] and for evaluation of terms we can use finite tree automata [5]. In our approach we have one problem in using these automata, our terms are typed and we have to extend the notion of automata to involve typing. To introduce types into finite deterministic sequential automata we follow the approach defined in [3, 4]. The authors define typed finite deterministic sequential automata (*TA*) for recognizing languages with structure. Input symbols of *TA* not only follow each other but bring some additional information that is appended to states of *TA* and so it is able to analyze the internal structure of input string. The original aim of *TA* was the grammatical inference in learning formal grammars for uknown languages from given data.

Our approach is quite different, but this method seems to be appropriate for using *TA* for type-checking and analyzing well-formedness of terms of our type theory. We assume that the additional information brought by input symbols is the knowledge that input symbols are typed. As input alphabet we use typed variables, operation names from a signature Σ and as auxiliary symbols the brackets. Left brackets are dummy symbols for *TA*, right brackets inform about finishing the analysis of a subterm. A *TA* has to check whether the input string is a well-formed term, i.e. it is constructed accordingly to term construction rules and is well-typed. In such a case *TA* terminates its work in a final state that is of expected type. But, e.g. the terms $plus)x$ and $plus(b, n)$ with $b : bool, n : nat$ are not well-formed terms and there is no *TA* accepting them.

Let $\Sigma = (T, F)$ be a signature as above. We define the set $\bar{F}$ containing only names of operations from Σ. The input alphabet of *TA* is $\bar{\Sigma} = Var \cup \bar{F} \cup \{(,)\}$.

A *typed finite deterministic sequential automaton* is a 7-tuple

$$TA = (\bar{\Sigma}, Q, q_1, Q_{final}, \delta, \bar{T}, type)$$

where

- $\bar{\Sigma}$ is an input alphabet defined above;
- Q is a finite set of states, every state has a type assigned by a function $type : Q \to \bar{T}$
- $\bar{T}$ is a set of Church's types together with σ_{init}, the type of initial state;

• $\delta\colon Q \times \bar{\Sigma} \to Q$ is a *transition (next state) function.*

A language $\mathcal{L}(TA)$ recognized by TA is a set of all typed terms $\Gamma \vdash t : \tau$ over a signature Σ such that $\delta(q_1, t) = q$, where $q \in Q_{final}$ and $type(q) = \tau$.

Example 3.1. A typed term $b : bool, x : int, n : nat \vdash if(b, exp(x, n), x) : int$ can be recognized by the following typed automaton *TA*:

• the input alphabet is $\bar{\Sigma} = \{b, x, n\} \cup \{if, exp\} \cup \{(,)\}$;
• the set of states is $Q = \{q_1, \ldots, q_9\}$;
• the set of final states is the singleton $Q = \{q_9\}$;
• the transition function is defined by the following equations:

$$\begin{array}{llll} \delta(q_1, if(\ ) = q_2 & \delta(q_2, b) = q_3 & delta(q_3, exp(\ ) = q_6 & \delta(q_6, x) = q_7 \\ \delta(q_7, n) = q_8 & \delta(q_8,)\ ) = q_4 & \delta(q_4, x) = q_5 & \delta(q_5,)\ ) = q_9 \end{array}$$

• and the typing function is defined as follows:

$$\begin{array}{ll} type(q_1) = \sigma_{init} & type(q_6) = int \times nat \to int \\ type(q_2) = bool \times int \times int \to int & type(q_7) = int \\ type(q_3) = bool & type(q_8) = nat \\ type(q_4) = int & type(q_9) = int \\ type(q_5) = int & \end{array}$$

This *TA* can be illustrated in Figure 1.

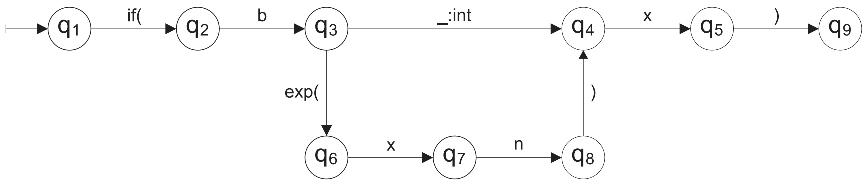

Figure 1: Typed sequential automaton

3.1. From type theory to typed sequential automata

Every TA can be depicted in the category **Set** consisting of sets as category objects and functions between them as category morphisms [1]. The category **Set** is biccc with singletons $\{*\}$ as terminal objects and the empty set as the initial object.

Let $Cl(\Sigma)$ be a classifying category over a signature Σ and $\Gamma \vdash t : \tau$ be a term. We assign to t a typed automaton in **Set** as follows. Because category objects in **Set** are all sets, the sets $\bar{T}$ and $\bar{\Sigma}$ are **Set**-objects. We take as the set Q the least set of states of types from $\bar{T}$. Because **Set** is biccc, there exists the product object $Q \times \bar{\Sigma}$. We define the transition function δ in **Set**

$$\delta : Q \times \bar{\Sigma} \to Q$$

with the property

$$\delta(q_1, t) = q, \quad \text{and} \quad \mathit{type}(q) = \tau.$$

Then we take as the final state a singleton $Q_{final} = \{q\}$. It is easy to define a function *type* because the input symbols deliver types to states. As the initialization function of TA we take a function $\{*\} \to Q$ from a terminal object in **Set** that provides the initial state $q_1 : \sigma_{init}$.

If $\Gamma \vdash t : \tau$ is a well-formed term in our type theory over a signature Σ, we can always find a typed automaton accepting it. The proof follows straightforward from the construction. We can show such typed automaton in **Set** in the diagram:

$$\begin{tikzcd}
 & & Q \times \bar{\Sigma} \arrow[dl, "\delta"'] \\
\{*\} \arrow[r] & Q \arrow[dr, "type"] & \\
 & & \bar{T}
\end{tikzcd}$$

4. Typed tree automata

A tree is a frequently used data structure in computer science. Every term t can be viewed as a finite labeled tree such that the leaves are variables or constants and internal nodes are labeled with function symbols with positive arity. From an internal node u lead $\mathit{arity}(u)$ edges. For our purposes we extend the notion of labeled trees with types. Every node in a typed tree has associated a type.

Example 4.1. Our typed term from previous example $b : \mathit{bool}, x : \mathit{int}, n : \mathit{nat} \vdash \mathit{if}(b, \mathit{exp}(x, n), x) : \mathit{int}$ can be considered as typed labeled tree in Figure 2:

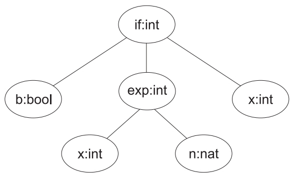

Figure 2: Example of typed tree

Tree automata [5] are devices that handle labeled trees analogously as sequential automata handle sequences of input symbols. In contrast to sequential automata which work from left to right, tree automata work bottom-up, from the leaves to the root. We denote by $\mathit{Tree}(\Gamma) = \bigcup_{\sigma \in \bar{T}} \mathit{Tree}_\sigma(\Gamma)$ the set of all trees of types from $\bar{T}$ in which can occur only variables declared in a type context Γ. We extend the notion of finite deterministic tree automata with types similarly as in the case of sequential case. States of a tree automaton are associated to every node of the tree. The labeling of each node is defined by transition function and the typed state of the root determines whether the tree belongs to the tree language or not. In the following we denote a labeled tree for a typed term $\Gamma \vdash t : \tau$ by t if it does not lead to confusion.

A *typed finite deterministic tree automaton* (*TTA*) is a structure

$$TTA = (Q, \bar{\Sigma}, \delta, Q_{final}, \Gamma, \lambda)$$

where

- $\bar{\Sigma}$ is an input alphabet defined as in the case of typed sequential automata;
- $Q = \bigcup_{\sigma \in \bar{T}} Q_\sigma$ is the finite set of typed states over $\bar{T}$;
- $Q_{final} \subseteq Q$ is a set of final states;
- $\delta : \bar{F} \times Q^* \to Q$ is a transition function. If $f : \sigma_1 \dots \sigma_n \to \tau$ is a function symbol from F then

$$\delta(f, q_1 \dots q_n) = q_{fin},$$

where $q_{fin} \in Q_{final}$, $type(q_i) = \sigma_i$, for $i = 1, \dots, n$ and $type(q_{fin}) = \tau$;

- $\Gamma = (v_1 : \sigma_1, \dots, v_n : \sigma_n)$ is a type context consisting of variable declarations occuring in a tree;
- $\lambda : \Gamma \to Q$ is an initialization mapping assigning to variables their typed states, i.e. values of type representations. For instance, if $x : \sigma$ then $\lambda(x) = a$, $a \in [\![\sigma]\!]$.

We extend a transition function to handle typed trees. We define a function $\delta' : \bar{F} \times (Tree_\sigma(\Gamma))^*_{\sigma \in \bar{T}} \to Q$ recursively as follows:

- if $arity(c) = \varepsilon$ then $\delta'(c) = \delta(c)$;
- if $arity(f) = type(\delta'(t_1), \dots, \delta'(t_n))$ then $\delta'(f(t_1, \dots, t_n)) = \delta(f, \delta'(t_1), \dots, \delta'(t_n))$.

We say that *TTA* accepts a tree t if and only if $\delta'(t) \in Q_{final}$. The language $\mathcal{L}(TTA)$ recognized by TTA is the set of all trees accepted by the automaton, it is a regular tree language.

Let $Trees_\tau(\Gamma)$ be a set of all trees of type τ over Γ. For a function symbol $f : \sigma_1 \dots \sigma_n \to \tau$ we can define a *tree construction function*

$$\varphi_f : \prod_{i=1}^{n} Tree_{\sigma_i}(\Gamma) \to Tree_\tau(\Gamma).$$

If $t_i \in Tree_{\sigma_i}(\Gamma)$, for $i = 1, \dots, n$ are trees then

$$\varphi_f(t_1, \dots, t_n) = t.$$

Computation mappings of typed trees can be defined by

$$\rho_f : (Tree_\tau(\Gamma), \varphi_f) \to (Q, \{\delta'_f\})$$

for every function symbol $f : \sigma_1 \dots \sigma_n \to \tau$. These mappings extend the initialization mapping λ.

Example 4.2. Let $b : bool, x : int, n : nat \vdash if(b, exp(x, n), x) : int$ be a typed term from previous examples. A possible *TTA* accepting this tree can be defined as follows. The set Q contains states of types from $T = \{bool, int, nat\}$, type context is $\Gamma = (b : bool, x : int, n : nat)$ and let the initialization mapping be defined by

$$\lambda(b) = true \quad \lambda(x) = -2 \quad \lambda(n) = 3$$

The computation sequence defined by ρ_{if} is in Figure 3. We see that $q_{fin} = -8$

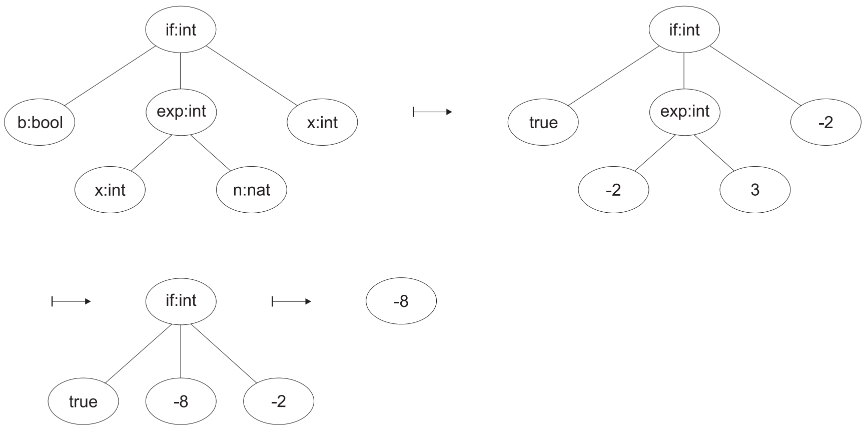

Figure 3: Computation sequence

and $type(q_{fin}) = int$.

4.1. Typed tree automata in Set

Similarly as in the case of typed sequential automata a typed tree automata can be depicted in the category **Set** as it is illustrated in Figure 4.

$$\begin{tikzcd}
& \Gamma \arrow[dd, "\lambda"] & \\
Q_{\sigma_1} & & \\
\vdots & \prod_{\sigma_i \in \Gamma} Q_{\sigma_i} \arrow[ul] \arrow[dl] \arrow[r, "\delta"] & Q_T \\
Q_{\sigma_n} & &
\end{tikzcd}$$

Figure 4: *TTA* in the category **Set**

5. Conclusion

In our paper we described Church's type theory over a signature Σ enclosed in the classifying category $Cl(\Sigma)$. We extended the sequential automata with types and we got a useful mechanism for checking well-formedness of terms. Similarly we extended tree automata with types and the resulting *TTA* can be used for evaluation of typed terms of our type theory. To incorporate typed automata into the frame of our mathematical theory of programming we defined a relation between type theory and typed automata in the category **Set**.

Acknowledgements.

References

[1] Adámek, J., Trnková, V., Automata and algebras in categories, *Kluver Academic Publishers*, Dordrecht, (1989).

[2] Barr, M., Wells, Ch., Category theory for computing science, *Prentice-Hall*, (1990).

[3] Bernard, M., Higuera, C., GIFT: Grammatical inference for terms, *Proc. Conf. d'Apprentissage CAP99*, Paris, (1999).

[4] Kermorvant, Ch., Higuera, C., Dupont, P., Learning typed automata from automatically labeled data, *Journal Electronique d'Intelligence Artificielle*, Vol. 6, No. 45, (2004).

[5] Comon, H. et al, TATA - Tree Automata techniques and applications, `www.grappa.univ-lille3.fr/tata/`

[6] Novitzká, V., Church's types in logical reasoning on programming, *Acta Electronica et Informatica*, Vol. 6, No. 2, Košice, (2006), 27–31.

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

[8] Tejfel, M., Horváth, Z., Kozsik, T., Extending the Sparkle Core language with object abstraction, *Acta Cybernetica*, Vol. 17., (2005), 419–445.

[9] Zólyomi, I., Porkoláb, Z., Kozsik, T., An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming, *Generative Programming and Component Engineering LNCS*, Vol. 2830, (2003), 209–227.

Valerie Novitzká, Daniel Mihályi, Viliam Slodičák
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University Košice
Letná 9, 041 20 Košice, Slovakia