How to apply linear logic in coalgebraical approach of computing

Viliam Slodičák, Pavol Macko
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 04200 Košice, Slovak Republic
{viliam.slodicak, pavol.macko}@tuke.sk

Abstract. *Linear logic provides a logical perspective on computational issues such as control of resources and order of evaluation. The most important feature of linear logic is that formulas are considered as actions. While classical logic treats the sentences that are always true or false, in linear logic it depends on an internal state of a dynamic system. Curry-Howard correspondence is a correspondence between logic and computing in informatics. In this contribution we present two ways of computations which correctness we prove by Curry-Howard correspondence. We show a standard way and a new way of computing based on hylomorphism by using coalgebras which is an alternative method.*

Keywords. duality, hylomorphism, linear logic, Curry-Howard correspondence

1 Introduction

Linear logic provides a logical perspective on computational issues such as control of resources and order of evaluation. In classical logic treats the sentences that are always true or false; but in linear logic the truth value depends on an internal state of a dynamic system. We showed in [9] a new way of computing factorial based on hylomorphism by using coalgebras. Because of the checking, the correctness of the program is the most important phase of transformation into logical formulae. In this contribution we present correctness of this computing by Curry-Howard correspondence.

2 Basic notions

We start our approach with the well-known notion of universal algebra: many-typed signature (the *signature* in the following text). A many-typed signature $\Sigma = (T, \mathcal{F})$ consists of a finite set T of the basic types needed for a problem solution denoted by symbols $\sigma, \tau \ldots$ and of a finite set $\mathcal{F}$ of function symbols. Each function symbol $f \in \mathcal{F}$ is of the form $f : \sigma_1, \ldots, \sigma_n \to \sigma_{n+1}$ for some natural n. Generally, we distinct in a signature the constructor operations which tell us how to generate data elements; the destructor operations, also called observers or transition functions that tell us what we can observe about data elements; and the derived operations that can be defined inductively or coinductively. If the operation f has been defined inductively, the value of f is defined on all constructors. In a coinductive definition of f the values of all destructors on each outcome $f(x)$ have been defined.

2.1 Category theory

Algebraic and coalgebraic concepts are based on category theory. A category $\mathscr{C}$ is mathematical structure consisting of objects, e.g. $A, B, \ldots$ and morphisms of the form $f : A \to B$ between them. Every object has the identity morphism and morphisms are composable. Morphisms between categories are called functors, e.g. a functor $F : \mathscr{C} \to \mathscr{D}$ from a category $\mathscr{C}$ into a category $\mathscr{D}$ which preserves the structure.

2.2 Linear Logic

Girard's linear logic [5] has offered great promise, as a formalism particularly well-suited to serve at the interface between logic and computer science. By using the Curry-Howard correspondence, propositions of linear logic are interpreted as types. This paradigm has been a cornerstone of new approach concerning connections between intuitionistic logic, functional programming and category theory [3]. We consider here intuitionistic linear logic because it is very suitable for describing of the program execution. Precisely, reduction of linear terms corresponding to proofs in intuitionistic linear logic can be regarded as a computation of programs [9]. The interpretation in linear logic is of hypotheses as resources: every hypothesis must be consumed exactly once in a proof. Its the most important feature is that formulae are considered as actions. That differs from usual logics where the governing judgement is of truth, which may be freely used as many times as necessary. Linear logic uses two conjunctions: multiplicative $\varphi \otimes \psi$ expressing that both actions will be performed; and additive one $\varphi \& \psi$ expressing that only one of two actions will be performed and we shall decide which one. Intuitionistic linear logic uses additive disjunction $\varphi \oplus \psi$ which expresses that only one of two actions will be performed but we cannot decide which one.

2.3 Curry-Howard correspondence

The Curry-Howard isomorphism is a correspondence between systems of mathematical logic and programming languages. It is the relationship between computer programs and proofs in constructive logic and it forms the *proofs-as-programs* and *formulae-as-types* paradigms. The concept was formulated by the mathematician H. Curry and logician W. A. Howard.

The *rôle* of the computer program is carrying on the instructions under whose the computer system is to perform some required computations. We consider programming as a logical reasoning over axiomatized mathematical theories needed for a given solved problem. A program is intuitively understood as data structures together with algorithms [7]. Data structures are always typed and operations between them can be regarded as algorithms. The results of computations are obtained by evaluation of typed terms. Due to a connection between linear logic and type theory [11], we are able to consider types as propositions and proofs as programs, resp. Then we are able to consider the program as a logical deduction within linear logical system. Thus computation of any resource-oriented program is some form of goal-oriented searching the proof in linear logic. This approach also keeps us away from potentional problems in the verification of programs.

3 Algebras and Coalgebras

The essential idea of the behavioral theory is to determine the relation between internal states and their observable properties. The internal states are often hidden. There are introduced many formal structures to capture the state-based dynamics, e.g. automata, transition systems, Petri nets, etc. Horst Reichel firstly introduced the notion of behavior in the algebraic specifications [8]. The basic idea was to disengage types in a specification into visible and hidden ones. Hidden types capture states and they are not directly accessible. The execution of a computer program causes a generation of some behavior that can be observed typically as a computer's input and output [6]. The observation of program behavior can be formularized by using the coalgebras. A program is considered as an element of the initial algebra arising from the used programming language. In other words it is an inductively defined set P of terms [7] which forms a suitable algebra $F(P) \to P$ where F is an endofunctor constructed over the signature. Then data type is completely determined by its constructors, algebraic operations, going into data type. Each language construct corresponds to certain dynamics captured in coalgebras. The behavior of programs is described by the final coalgebra $P \to G(P)$ where the functor G captures the kind of behavior that can be observed. Shortly, generated computer behavior amounts to the repeated evaluation of a (coinductively defined) coalgebraic structure on an algebra of terms. The state can be observed via the visible values and can be modified. In coalgebra it is realised using destructor operations pointing out of the structure. Thus coalgebraic behavior is generated by an algebraic program [7, 9]. Therefore the algebras are used for constructing basic struc-

tures used in computer programs and coalgebras act on the state space of computer describing what can be observed externally. For expressing the relations we use categories. Because the objects of category can be arbitrary structures, categories are useful in computer science, where we often use more complex structures not expressible by sets [2].

Algebras and coalgebras are considered as dual structures. Usually we treat them in categories [10]. We use special kind of algebras and coalgebras - an initial algebra and a final coalgebra, resp. It holds for initial algebra, that there exists the unique morphism from the initial algebra into any algebra. This morphism is called the *catamorphism*. Dually, there exists final coalgebra, for which holds, that from any coalgebra exists unique morphism into final coalgebra, called *anamorphism* [1]. Composition of those morphisms is an another morphism which is called the *hylomrphism*. We apply it in the alternative way of the factorial computation.

3.1 Initial algebras

Let F be an endofunctor from $\mathscr{C}$ to $\mathscr{C}$. An algebra with the signature F (or an F-algebra for short) is a pair (A, α) where A called the carrier is an object and the algebra structure $\alpha : FA \to A$ is a morphism in $\mathscr{C}$. For any two F-algebras (A, α) and (C, γ), a morphism $f : A \to C$ is said to be a homomorphism of F-algebras from (A, α) to (C, γ), so the following diagram at fig. 1 commutes.

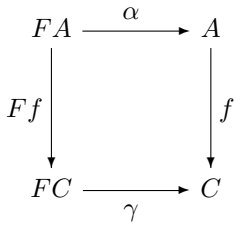

Figure 1: Diagram of algebras

It follows from the diagram at Fig. 1 that it holds the equality $\alpha \circ f = Ff \circ \gamma$. An F-algebra is said to be an initial F-algebra if it is an initial object of the category $\mathscr{A}lg(F)$ of F-algebras. The existence of initial algebra of the endofunctor is constrained by the fact that initial algebras, when they exist, must fulfil some important properties: they are unique up to isomorphism and the initial algebra has an inverse. It follows from the first property that there exists at most one initial F-algebra. Because from the initial F-algebra exists unique homomorphism to every F-algebra, the initial T-algebra is the initial object in the category $\mathscr{A}lg$. The second property was proven by J. Lambek and it says that the initial F-algebra is the least fixed point of the endofunctor F.
The initiality provides a general framework for induction and recursion. Given a functor F, the existence of the initial F-algebra $(\mu F, in_F)$ means that for any F-algebra (A, α) there exists a unique homomorphism of algebras from $(\mu F, in_F)$ into (A, α). Following [4], we denote this homomorphism by $(cata\ \alpha)_F$, so $(cata\ \alpha)_F : \mu F \to A$ is characterized by the universal property [12]:

$$f \circ in_F = \alpha \circ Ff \quad \Leftrightarrow \quad f = (cata\ \alpha)_F.$$

The type information is summarized in the following commutative diagram at Fig. 2.

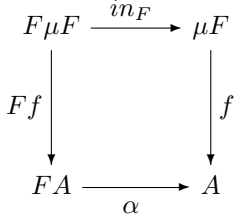

Figure 2: Diagram of initial algebra and catamorphism

Morphisms of the form $(cata\ \alpha)_F$ are called *catamorphisms*; the structure $(cata\ (\_))_F$ is an iterator.

3.2 Final coalgebras

Coalgebras are dual structures to algebras. Let F be an endofunctor from $\mathscr{C}$ to $\mathscr{C}$. A coalgebra with the signature F (an F-coalgebra for short) is a pair (U, φ), where U called the state space is an object and $\varphi : U \to FU$ called the coalgebra structure (or coalgebra dynamics) is a morphism in $\mathscr{C}$.

For any two F-coalgebras (T, ψ) and (U, φ), a morphism $f : T \to U$ is said to be a homomorphism from (T, ψ) to (U, φ) between F-coalgebras, so the following diagram at Fig. 3 commutes.

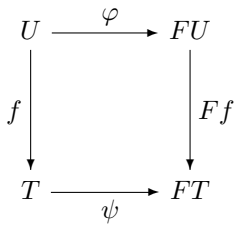

Figure 3: Diagram of coalgebras

and it holds the equality $\varphi \circ Ff = f \circ \psi$.

The F-coalgebras and the homomorphisms between them form a category. The category $\mathscr{C}oalg(F)$ is the category whose objects are the F-coalgebras and morphisms are the homomorphisms between them. Composition and identities are inherited from $\mathscr{C}$. An F-coalgebra is said to be a final F-coalgebra if it is the final object of the category $\mathscr{C}oalg(F)$.

The existence of the final F-coalgebra $(\nu F, out_F)$ means that for any F-coalgebra (U, φ) there exists a unique homomorphism of coalgebras from (U, φ) to $(\nu F, out_F)$. This homomorphism is usually denoted by $(ana\ \varphi)_F$, so $(ana\ \varphi)_F : U \to \nu F$ is characterized by the universal property [12]:

$$out_F \circ f = Ff \circ \varphi \quad \Leftrightarrow \quad f = (ana\ \varphi)_F.$$

The type information is summarized in the following diagram at Fig. 4.

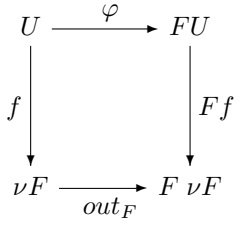

Figure 4: Diagram of final coalgebra and anamorphism

Morphisms of the form $(ana\ \varphi)_F$ are called *anamorphisms* and the structure of $(ana\ (\_))_F$ is a coiterator.

3.3 Recursive coalgebra

The concept of the recursive coalgebra, i.e. a coalgebra which has a unique coalgebra-to-algebra morphism into every algebra is important for the formulation of the relation between coalgebras and algebras in one category. Recursive coalgebras extend that universal property beyond the initial algebra considered as coalgebra [1].

Let $F : \mathscr{C} \to \mathscr{C}$ be an endofunctor. A coalgebra (U, φ) is called recursive if for every algebra (A, α) there exists a unique coalgebra-to-algebra morphism $f : U \to A$ at Fig. 5.

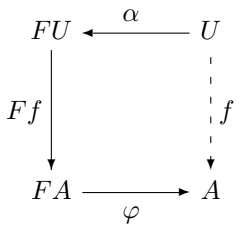

Figure 5: Diagram of recursive coalgebra

So it holds the equality $f = \alpha \circ Ff \circ \varphi$.

3.4 Hylomorphism

The hylomorphism recursion pattern was firstly defined in [4]. Given an F-coalgebra $\varphi : U \to FU$ and an F-algebra $\alpha : FA \to A$, the hylomorphism denoted by $hylo(\alpha, \varphi)_F$ is the least arrow $f : U \to A$ that makes the following diagram at Fig. 6 commute.

Moreover, the hylomorphism is a composition of an anamorphism with a catamorphism [12]:

$$hylo(\alpha, \varphi)_F = (cata\ \alpha)_F \circ (ana\ \varphi)_F.$$

The hylomorphism captures general recursion by producing the complex data structure and then processing it.

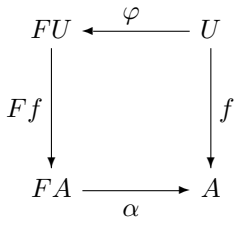

Figure 6: Diagram of hylomorphism

4 The computation and logical proof

Using the Curry-Howard correspondence we are able to consider proofs as programs and execution of a program as a logical deduction in considered logical system. The first step in the design solution is constructing the type theory that we will use for a given problem. The types together with operations over them we enclose into a many-typed signature $\Sigma = (T, \mathcal{F})$.

4.1 Traditional computation

Traditional mathematical way of the factorial computing is as follows.

$$fact\,(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ n * fact\,(pred\ n) & \text{otherwise} \end{cases}$$

The type theory for a given problem we construct as a signature $\Sigma = (T, \mathcal{F})$. Set of types contains the types for numerical values, tuples of values and the type of truth values Ω.

$$T = \{nat, nat \times nat, \Omega\}\,.$$

Set of function symbols contains operations over those types in T used for the calculation of factorial.

$$\begin{aligned} \mathcal{F} \quad = \quad & \{pred : nat \to nat, \\ & \quad =: nat, nat \to \Omega \\ & mult : nat, nat \to nat \\ & zero :\to nat \\ & one :\to nat\} \end{aligned}$$

Term for factorial in the type theory has the form

$$\begin{array}{l} n : nat \vdash \\ \text{if } (n = 0) \vee (n = 1) \text{ then } 1 \text{ else } n * fakt\,(pred\ n)\,. \end{array}$$

Corresponding formula in linear logic for the given term is

$$(\varphi \multimap \psi_1)\,\&\big(\varphi^{\perp} \multimap \psi_2\big)$$

where

$$\begin{array}{l} \varphi : (n = 0\ \&\ n = 1) \\ \psi_1 : fact = 1 \\ \psi_2 : fact = n * fact\,(pred\ n) \end{array}$$

So the formula is

$$\begin{array}{l} ((n = 0\ \&\ n = 1) \multimap fact = 1)\ \& \\ \quad ((n > 1) \multimap fact = n * fact\,(pred\ n)) \end{array}$$

Now we are able to construct the logical proof for a given formula. The fragment of proof is depicted at Fig. 7.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Theta, n = 0 \vdash fact = 1}}{\Theta, (n = 0)\&(n = 1) \vdash fact = 1}}{\Theta \vdash \varphi \multimap \psi_1}\multimap I \quad \dfrac{\dfrac{\dfrac{}{\Theta, n = 1 \vdash fact = 1}}{\Theta, (n = 0)\&(n = 1) \vdash fact = 1}}{\Theta \vdash \varphi \multimap \psi_1}\multimap I}{\Theta \vdash (\varphi \multimap \psi_1)\,\&\big(\varphi^{\perp} \multimap \psi_2\big)}\,\&I \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Theta, (n > 1) \vdash fact = n * fact(pred\ n)}}{\Theta \vdash \varphi^{\perp} \multimap \psi_2}\multimap I}{\Theta \vdash \varphi^{\perp} \multimap \psi_2}\multimap I}{\Theta \vdash (\varphi \multimap \psi_1)\,\&\big(\varphi^{\perp} \multimap \psi_2\big)}\,\&I}{\cdots}$$

Figure 7: Proof of formula expressing standard factorial computation

Finally, the corresponding program in *OCaml* is

```
let rec fact n =
if (n==0) or (n==1) then 1
else n*fact(pred n);;
```

4.2 Alternative method for the factorial calculation

We show alternative method of the factorial calculation based on algebras and coalgebras. The signature consists of a finite set of the basic types

$$T = \{int, intList, \Omega\}$$

and of a set of function symbols:

$$\begin{aligned}\mathcal{F} = \{ \quad & ==: intList, intList \to \Omega, \\ & =: int, int \to \Omega, \\ & join : int, intList \to intList, \\ & * : int, int \to int, \\ & pred : int \to int, \\ & head : intList \to int, \\ & tail : intList \to intList \ \}\end{aligned}$$

For our alternative method for computation of the factorial we need terms, which represent catamorphism and anamorphism. Our function $fact(n)$ is based on hylomorphism. Function $fact$ consist of composition two functions. Listed functions are named by morphisms which are representing, namely: *cata* and *ana*, resp.

4.2.1 Anamorphism

An anamorphism usually represents a corecursive function that starts with a single input (here *nat*) and it returns more complex output, here a wide list (*natList*). The function *ana* it is of type $nat \to natList$.
The definition of function *ana* is as follows:

$$\begin{aligned}& ana(n) = \\ & if\ (n = 0)\ then\ ana = emptyList \\ & elseif\ (n = 1)\ then\ ana = [1] \\ & else\ ana = join(n, ana(pred\ n))\end{aligned}$$

Typed term that represents the function *ana* has the following form:

$$\begin{aligned}& n : nat \vdash if\ (n = 0)\ then\ \varepsilon \\ & elseif\ (n = 1)\ then\ [1]\ else\ join(n, ana(pred\ n))\end{aligned}$$

Formula representing the function $ana(n)$ is:

$$(\varphi_1 \multimap \psi_1)\ \&\ (\varphi_2 \multimap \psi_2)\ \&\ ((\varphi_1^{\perp} \otimes \varphi_2^{\perp}) \multimap \psi_3)$$

where

$$\begin{aligned}& \varphi_1 : (n = 0) \\ & \varphi_2 : (n = 1) \\ & \psi_1 : ana = \varepsilon \\ & \psi_2 : ana = [1] \\ & \psi_3 : ana = join(n, ana(pred\ n))\end{aligned}$$

4.2.2 Catamorphism

By applying the catamorphism in the informatics, we get a recursive function that starts with a list (here *natList*) and it returns a single numerical output (here *nat*). The function *cata* is of type $natList \to nat$.

Definiton of this function:

$$\begin{aligned}& cata(list) = \\ & if\ (list = emptyList)\ then\ cata = 1 \\ & else\ cata = head(list) * cata(tail(list))\end{aligned}$$

Typed term that represents the function *cata* has the following form:

$$\begin{aligned}& l : natList \vdash if\ (list == \varepsilon)\ then\ 1 \\ & else\ head(list)\ *\ cata(tail(list))\end{aligned}$$

Formula representing the function $cata(l)$ is:

$$(\theta \multimap \alpha) \& (\theta^{\perp} \multimap \beta)$$

where

$$\begin{aligned}& \theta : list = emptyList \\ & \alpha : cata = 1 \\ & \beta : cata = head(list) * cata(tail(list))\end{aligned}$$

4.2.3 Function for calculating the factorial

The composition of functions *ana* a *cata* creates a function $fact(n)$ for the factorial computation. The function generates a list of natural numbers by increments from 1 to number n, and simultaneously the list is eliminated by the multiplication operation between elements of the list. The function is of type $nat \to natList \to nat$.
Definiton of the function $fact(n)$:

$$\begin{aligned}& fact(n) = cata(ana(n)) = \\ & if\ (ana(n) == emptyList)\ then\ fact = 1 \\ & else\ fact = n * cata(ana(predn))\end{aligned}$$

Typed term that represents the function *cata* has the following form:

$$n : nat \vdash if\ (ana(n) == \varepsilon)\ then\ 1 \\ else\ n\ *\ cata(ana(pred\ n))$$

Formula representing the function $fact(n)$ is:

$$((\varphi_1 \multimap \psi_1) \multimap \alpha)\ \&\ ((\varphi_2 \multimap \psi_2) \multimap \alpha)\ \& \\ \&\ (((\varphi_1^{\perp} \otimes \varphi_2^{\perp}) \multimap \psi_3) \multimap \beta)$$

4.2.4 The proof

The logical proof of the given formula in the section 4.2.3 is at Fig. 8.

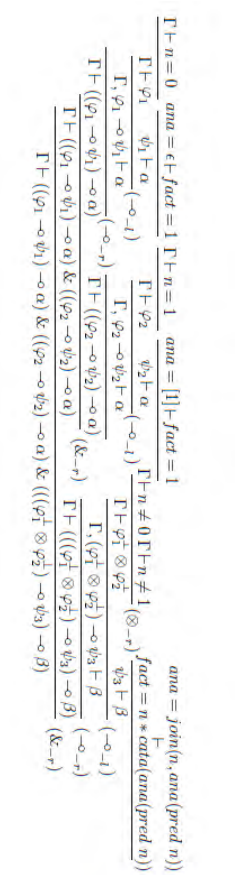

Figure 8: Proof of formula expressing alternative factorial computation

When the formula is proven it means that our program is correct and it does not need any verification.

5 Implementation in *OCaml*

In this section we show the implementation of our method for the factorial calculating. We use the new functional language *OCaml*.

5.1 The function *ana*

This function is defined as follows: if the argument of the function *ana* is 0 then it returns an empty list. If the argument is 1 then *ana* generates a list containing only 1 as item. Otherwise, *ana* generates a list with new element appended. The implementation of the function $ana(n)$ is:

```
let rec ana n =
match n with
| 0 -> []
| 1 -> [1]
| n -> n :: ana (n-1);;
```

5.2 The function *cata*

This function takes as an argument a list of factors of the type *nat* and returns the result of multiplicative operations over the list by multiplicating the values from the input list. The result of the function is an element of the type *nat* which is the result of multiplication of elements in the list. The implementation of the function `cata(l)` is:

```
let rec cata list =
match list with
| [] -> 1
| head :: tail -> head * (cata tail);;
```

5.3 The function *fact*

Composition of two function $cata \circ ana$ is written in programming language *OCaml* as $cata(ana(n))$. The definition of this hylomorphism function $fact(n)$ is as follows:

```
let fact n =
cata (ana n);;
```

Execution of this function with input value 4 is:

```
# fact 4;;
- : int = 24
```

Illustration of this example step by step:

```
fact 4 =
       cata (ana 4) =
    4  cata (ana 3) =
   12  cata (ana 2) =
   24  cata (ana 1) =
   24 id =
   24
```

It is seen that our alternative method of programming based on hylomorphism provides the expected results. Because it has been proven in linear logic as a formula representing our function in the Curry-Howard correspondence, we can say tgat our function is correct.

6 Conclusion

We presented an alternative way of the factorial calculation which is based on the algebraic and coalgebraic structures expressed in categories. These structures provide the computation which we proved with the Curry-Howard correspondence and we constructed the logical proof of the appropriate formulae in linear logic. Our next goal will be the extension of this approach in other categorical structures based on monads and comonads and to formulate the linear logic proofs in appropriate categorical structures.

Acknowledgment

This paper has been supported by project VEGA No. 1/0015/10 Principles and methods of semantics enrichment and adaptation of knowledge-based languages for automatic software development.

References

[1] Adámek, J., Lücke, D., and Milius, S. Recursive coalgebras of finitary functors. *ITA 41*, 4 (2007), 447–462.

[2] Barr, M., and Wells, C. *Category Theory for Computing Science*. Prentice Hall International, 1990. ISBN 0-13-120486-6.

[3] Blute, R., and Scott, P. *Category theory for linear logicians.* T.Erhard, J.-Y.Girard, P.Ruet (eds.): Linear Logic in Computer Science. London Mathematical Society Lecture Note Series, Cambridge Univ.Press, 2004.

[4] Fokkinga, M., and Meijer, E. Program calculation properties of continuous algebras. Tech. rep., CWI, Amsterdam, 1991. Technical Report CS-R9104.

[5] Girard, J. *Linear logic: Its syntax and semantics*. Cambridge University Press, 2003.

[6] Jacobs, B. Introduction to coalgebra. *Towards Mathematics of States and Observations (draft)* (2005).

[7] Novitzká, V., Mihályi, D., and Verbová, A. Coalgebras as models of systems behaviour. In *International Conference on Applied Electrical Engineering and Informatics, Greece, Athens* (2008), pp. 31–36.

[8] Reichel, H. Behavioural equivalence - a unifying concept for initial and final specification methods. In *3rd Hungarian Computer Science Conference* (1981), no. 3, Akadémia kiadó, pp. 27–39.

[9] Slodičák, V., and Macko, P. New approaches in functional programming using algebras and coalgebras. In *European Joint Conferrences on Theory and Practise of Software - ETAPS 2011* (March 2011), Universität des Saarlandes, Saarbrücken, Germany, pp. 13–23. ISBN 978-963-284-188-5.

[10] Slodičák, V., and Ľaľová, M. Some useful structures for categorical approach for program behavior. In *Proceedings of the 21st Central European Conference on Information and Intelligent Systems - CECIIS 2010* (September 2010), Faculty of Organization and Informatics, University of Zagreb, Varaždin, pp. 477–484. ISSN 1847-2001.

[11] Sørensen, M., and Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism*. University of Copenhagen a University of Warsaw, 1999.

[12] Uustalu, T., and Vene, V. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Vilnius: Informatica 10*, 1 (1999), 5–26.