UDC 164:004
Original Scientific Paper

The rôle of linear logic in coalgebraical approach of computing

Viliam Slodičák *viliam.slodicak@tuke.sk*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical university of Košice
Letná 9, 042 00 Košice
Slovak Republic

Pavol Macko *pavol.macko@tuke.sk*
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical university of Košice
Letná 9, 042 00 Košice
Slovak Republic

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 formulae 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 show 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. Our method of recursive and corecursive computations we apply in simple authentication system.

Keywords: anamorphism, catamorphism, hylomorphism, linear logic, Curry-Howard correspondence, authentication system

1. Introduction

An important problem in theoretical computer science is discovering logical foundations of programming languages. One of the most fruitful methods used to explore such logical foundations has been to utilize a fascinating relationship between various typed $lambda$-calculi, constructive logic and models from category theory. For example, intuitionistic logic corresponds to simply typed λ-calculus and to the cartesian closed category as the categorical model. Another important example is the higher-order intuitionistic logic [18], which corresponds to higher order typed λ-calculus and to the toposes as the categorical model [16]. 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 formulae are considered as actions. While classical logic treats the sentences that are always true or false, in linear logic the truth value depends on an internal state of a dynamic system. We showed in [19] a new way of computing factorial based on hylomorphism by using coalgebras. In this contribution we present correctness of this computing by linear logic and the Curry-Howard correspondence. Usually the Curry-Howard correspondence [21] is used as follows: At first we make the proof of the problem and then we develop the program. But Curry-Howard correspondence is bijective, so it applies to both ways. Here we show how to prove the correctness of the developed program. This opposite method is called *re-*

verse engineering. Because of checking the correctness of program is the most important phase of transformation into logical formulae. Then the proof is constructed in linear logic. Our recursive and corecursive methods of computation we apply in a model of simple authentication system.

2. Basic notions

We start our approach with the well-known notion from universal algebra: a 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. Every function symbol $f \in \mathcal{F}$ is of the form $f : \sigma_1, \ldots, \sigma_n \to \sigma_{n+1}$ for some natural number n. Using the type constructors on basic types from Σ we construct Church's types: product types, coproduct (sum) types and function types [14]. Generally, we distinct in a signature:

- constructor operations which tell us how to generate (algebraic) data elements;
- destructor operations, also called observers or transition functions, that tell us what we can observe about our data elements;
- 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, i.e. it takes inputs of types $\sigma_1, \ldots, \sigma_n$ and yields an output of a type σ_{n+1}.

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 $id_A : A \to A$ and morphisms are composable. 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. 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 [7] has offered great promise, as formalism particularly well-suited to serve at the interface between logic and computer science. Linear logic provides a logical perspective on computational issues such as control of resources and order of evaluation. 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 [5]. 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 [19].

The interpretation in linear logic is of hypotheses as resources: every hypothesis must be consumed exactly once in a proof. The most important feature of linear logic is that formulae are considered as actions. That differs from usual logic where the governing judgment is of truth, which may be freely used as many times as necessary. While classical and intuitionistic logic treat the sentences that are always true or false, in linear logic formulae describe actions and the truth values depend on an internal state of a dynamic system. For instance, a linear implication $\varphi \multimap \psi$ is causal, i.e. the action described by formula φ is a cause of the action described by ψ; the formula

φ does not hold after linear implication. 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 the two actions will be performed and we can decide which one. Intuitionistic linear logic uses additive disjunction $\varphi \oplus \psi$ which expresses that only one of the two actions will be performed but we cannot decide which one. Dual of multiplicative conjunction is multiplicative disjunction $\varphi \invamp \psi$ and is read as "par". We consider here intuitionistic linear logic (the logic without "par" operation) because we would like to use it to describe program execution.

2.3 Curry-Howard correspondence

The Curry-Howard isomorphism is a correspondence between systems of mathematical logic and programming languages. It is formulated as a relationship between the 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. Knowledge of the Curry-Howard correspondence has enabled the development of programming languages in which we can actually type up and then automatically check mathematical proofs. The computer will tell us if our proof has any errors.

The *rôle* of the computer program is carrying on the instructions under whose the computer system is to perform some required computations. A running program should provide us a desirable solution of a given problem. 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 [8]. 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 - a phenomenon of the Curry-Howard correspondence [21], 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. One of the main advantages of this approach is that it is helping to avoid eventual mistakes of correctness generated by implementation of a programming language. This approach also keeps us away from potential 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 [17]. 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 [9]. The observation of program behavior can be formulated by using the coalgebras. A program can be 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 [15]. This set forms a suitable algebra $F(P) \rightarrow P$ where F is an endofunctor constructed over the signature of the operations appointed to execution by a program. So a 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 $P \rightarrow G(P)$ 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 realized by using destructor operations pointing out of the structure. Thus coalgebraic behavior is generated by an algebraic program [19]. Therefore the algebras are used for constructing basic structures 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 [4].

Algebras and coalgebras are considered as dual structures. Usually we treat them in categories [4, 20]. 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 a new morphism which is called the *hylomorphism*. We apply it in the alternative way of the factorial computation.

3.1 Initial algebras and catamorphisms

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.

$$\begin{tikzcd}
FA \arrow[r, "\alpha"] \arrow[d, "Ff"'] & A \arrow[d, "f"] \\
FC \arrow[r, "\gamma"'] & C
\end{tikzcd}$$

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 fulfill the following important properties:

- they are unique up to isomorphism, therefore we write initial F-algebra as $u : FU \cong U$, and
- the initial algebra has an inverse $u^{-1} : U \to FU$

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 F-algebra is the initial object in the category $\mathscr{A}lg$. The second property was proven in [11] which means that the initial F-algebra is the least fixed point of the endofunctor F. Initial algebras are generalizations of the least fixed points of monotone functions, since they have unique maps into arbitrary F-algebra [3].

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 [6], we denote this homomorphism by $(cata\ \alpha)_F$, so $(cata\ \alpha)_F : \mu F \to A$ is characterized by the universal property [22]:

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

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

$$\begin{tikzcd}
F\mu F \arrow[r, "in_F"] \arrow[d, "Ff"'] & \mu F \arrow[d, "f"] \\
FA \arrow[r, "\alpha"'] & A
\end{tikzcd}$$

Figure 2: Diagram of initial algebra and catamorfism

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

3.2 Final coalgebras and anamorphisms

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

$$\begin{tikzcd}
U \arrow[r, "\varphi"] \arrow[d, "f"'] & FU \arrow[d, "Ff"] \\
T \arrow[r, "\psi"'] & FT
\end{tikzcd}$$

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 [22]:

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

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

$$\begin{tikzcd}
U \arrow[r, "\varphi"] \arrow[d, "f"'] & FU \arrow[d, "Ff"] \\
\nu F \arrow[r, "out_F"'] & F\,\nu F
\end{tikzcd}$$

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.

$$\begin{tikzcd}
FU \arrow[d, "Ff"'] & U \arrow[l, "\alpha"'] \arrow[d, dashed, "f"] \\
FA \arrow[r, "\varphi"'] & A
\end{tikzcd}$$

Figure 5: Diagram of recursive coalgebra

The Fig. 5 gives equality as follows:

$$f = \alpha \circ Ff \circ \varphi.$$

3.4 Hylomorphism

The hylomorphism recursion pattern was firstly defined in [6]. 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 [10].
Moreover, the hylomorphism is a composition of an anamorphism with a catamorphism [10]:

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

$$\begin{tikzcd}
FU \arrow[d, "Ff"'] & U \arrow[l, "\varphi"'] \arrow[d, "f"] \\
FA \arrow[r, "\alpha"'] & A
\end{tikzcd}$$

Figure 6: Diagram of hylomorphism

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

4. The computation and logical proof

By using the Curry-Howard correspondence we are able to consider proofs as programs and execution of a program as a logical deduction in a 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 well-know notion from the universal algebra, a many-typed signature $\Sigma = (T, \mathcal{F})$.

4.1 Traditional way of the factorial computation

Traditional mathematical way of the factorial computing is the following [13]:

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

The type theory for a given problem we construct as a signature $\Sigma = (T, \mathcal{F})$ (Fig. 7). 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.

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

Term for factorial in the type theory has the form

$$n : nat \vdash \text{if } (n = 0) \vee (n = 1) \text{ then } 1 \text{ else } n * fact\,(pred\,n)\,.$$

Corresponding formula in linear logic for the given term is

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

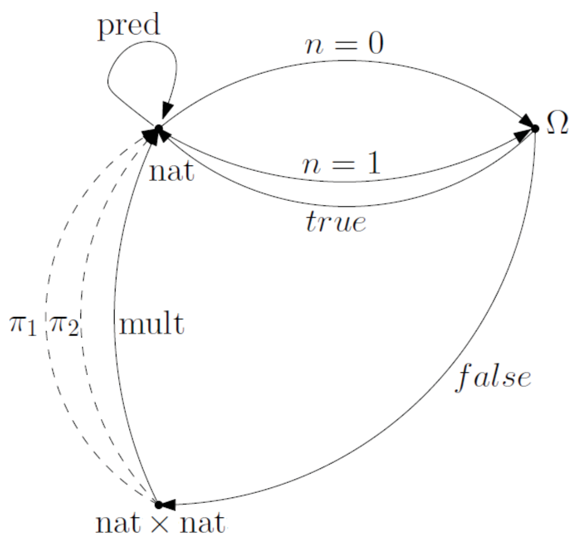

Figure 7: The many-typed signature for the factorial

where

$$\varphi : (n = 0 \,\&\, n = 1)\,; \quad \psi_1 : fact = 1; \quad \psi_2 : fact = n * fact\,(pred\; n)$$

So the formula is

$$((n = 0 \,\&\, n = 1) \multimap fact = 1) \;\; \& ((n > 1) \multimap fact = n * fact\,(pred\; n))\,.$$

Now we are able to construct the logical proof for a given formula. The fragment of proof is depicted at Fig. 8.

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

Figure 8: 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

Now we show alternative method of the factorial calculation. Our method is based on algebras and coalgebras. The signature (Fig. 9) consists of a finite set of the basic types

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

and of a set of function symbols:

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

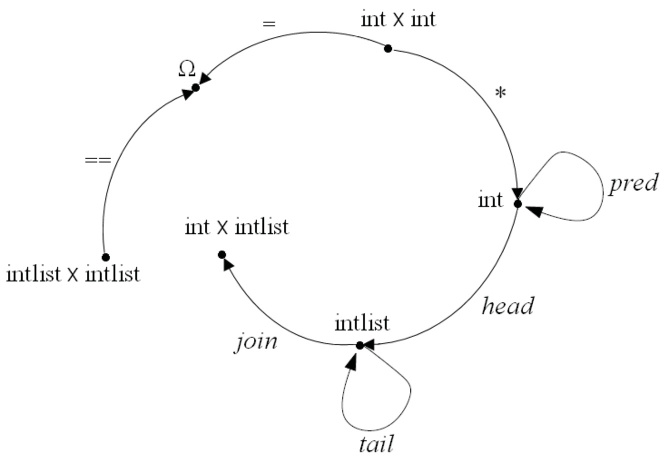

Figure 9: Signature for the alternative method of factorial computation

If the definition of types theories for the problem is fully, then we are able to formulate terms and by using the Curry-Howard correspondence we find a type to each formula. 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, which has been defined as a composition of cata and ana. Function $fact$ consist of composition two functions, too. Listed functions are named by morphisms which they 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 int) and it returns more complex output, here a wide list ($intList$). The function ana it is of type

$$int \to intList.$$

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:

$$n : nat \vdash if\ (n = 0)\ then\ \varepsilon\ elseif\ (n = 1)\ then\ [1]\ else\ join(n, ana(pred\ n))$$

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

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

4.2.2 Catamorphism

By applying the catamorphism in the informatics we get a recursive function that starts with a list (here $intList$) and it returns a single numerical output (here int). The function $cata$ is of type

$$intList \to int.$$

Definiton of this function:

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

Typed term that represents the function $cata$ has the following form

$$l : intList \vdash if\ (list == \varepsilon)\ then\ 1\ else\ head(list)\ *\ cata(tail(list))$$

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

$$(\theta \multimap \alpha)\&(\theta^\perp \multimap \beta),$$

where

$$\theta : list = emptyList; \quad \alpha : cata = 1; \quad \beta : cata = head(list) * cata(tail(list))$$

4.2.3 The 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 from 1 to n by incrementing; and simultaneously the list is eliminated by the multiplication operation between elements of the list. The function is of type

$$int \to intList \to int.$$

Definiton of the function $fact(n)$:

$$\begin{array}{ll} fact(n) = & cata(ana(n)) = \\ & if\ (ana(n) == emptyList)\ then\ fact = 1 \\ & else\ fact = n * cata(ana(pred\, n)) \end{array}$$

Typed term that represents the function $fact$ 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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\vdash n=0}{\Gamma\vdash\varphi_1} \quad \dfrac{ana=\epsilon\vdash fact=1}{\psi_1\vdash\alpha}}{\Gamma,\varphi_1\multimap\psi_1\vdash\alpha}(\multimap_{-l})}{\Gamma\vdash((\varphi_1\multimap\psi_1)\multimap\alpha)}(\multimap_{-r}) \quad \dfrac{\dfrac{\dfrac{\Gamma\vdash n=1}{\Gamma\vdash\varphi_2} \quad \dfrac{ana=[1]\vdash fact=1}{\psi_2\vdash\alpha}}{\Gamma,\varphi_2\multimap\psi_2\vdash\alpha}(\multimap_{-l})}{\Gamma\vdash((\varphi_2\multimap\psi_2)\multimap\alpha)}}{\Gamma\vdash((\varphi_1\multimap\psi_1)\multimap\alpha)\ \&\ ((\varphi_2\multimap\psi_2)\multimap\alpha)}(\&_{-r}) \quad \dfrac{\dfrac{\dfrac{\Gamma\vdash n\neq 0\ \Gamma\vdash n\neq 1}{\Gamma\vdash\varphi_1^{\perp}\otimes\varphi_2^{\perp}}(\otimes_{-r}) \quad \dfrac{\begin{array}{c}ana=join(n,ana(pred\ n))\\ \vdash \\ fact=n*cata(ana(pred\ n))\end{array}}{\psi_3\vdash\beta}}{\Gamma,(\varphi_1^{\perp}\otimes\varphi_2^{\perp})\multimap\psi_3\vdash\beta}(\multimap_{-l})}{\Gamma\vdash(((\varphi_1^{\perp}\otimes\varphi_2^{\perp})\multimap\psi_3)\multimap\beta)}(\multimap_{-r})}{\Gamma\vdash((\varphi_1\multimap\psi_1)\multimap\alpha)\ \&\ ((\varphi_2\multimap\psi_2)\multimap\alpha)\ \&\ (((\varphi_1^{\perp}\otimes\varphi_2^{\perp})\multimap\psi_3)\multimap\beta)}(\&_{-r})$$

Figure 10: Proof of formula expressing alternative factorial computation

5. The proof

The logical proof of the given formula in the section 4.2.3 is at Fig. 10.
When our formula is proven, it means that our program is correct and it does not need any verification.

6. Implementation in $OCaml$

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

6.1 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);;
```

6.2 The function $cata$

This function takes as an argument a list of factors of the type int and returns the result of multiplicative operations over the list by multiplication the values from the input list. The result of the function is an element of the type int 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);;
```

6.3 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 following

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

Now we show example of the evaluation of function $fact$ with input value 4. Execution of this function in OCaml:

```
# 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 by using the hylomorphism provides the expected results. Because it has been proven in linear logic as a formula representing our function, we can say that our function is correct.

7. Where do we go from here

We showed an alternative method for the factorial computation which is based on recursion and corecursion. Our idea is to use this method in real-life systems. We consider here very simple model of the authentication system where user is allowed to get in only if he/she is authorized to do it. There exist some other approaches of how to model a server implementation [2, 12]. In our method of authentication the user is not asked for input the password. The reason is that possible attacker in the network can catch the users' passwords. When through the network only a sequence of seeming the random numbers is sent, the attacker would not be able to recognize their meaning and to stole the enrolling data.

7.1 The model of system

Authentication system (Fig. 11) is based on the client-server process platform. The clients request from server an access to service. On the server side is a database containing the personal data of the users. Each user has the unique user identification number (*id*). When the client requests from server an access to service, he is confirmed whether the *id* is in database. If the user's id is in the database of users, then server generates two control numbers m and n by the following way:

$$m = 10 + det\begin{pmatrix} cs(id) & cs(hour) \\ cs(min) & cs(sec) \end{pmatrix} mod\, 21$$

$$n = 21 + det\begin{pmatrix} cs(id) & cs(hour) \\ cs(min) & cs(sec) \end{pmatrix} mod\, 31$$

where cs is the function of cipher sum, det is a determinant and $hour$, min and sec are the hour, minute and second of the time on the server when user requests an access to service from server. Here we have chosen the interval for the Fibonacci numbers from 10 to 20 for the first number and from 21 to 30 for the second number. This is because we would like to express numbers which are not very small for security and for usability of an algorithm also not very great. The numbers m and n can be sent to the user for instance by SMS message. User has his own simple application for calculating the response. After getting the data from server user inputs the numbers m and n to the user's application. The result for access is a string which contains m-th and n-th

Fibonacci number. Both Fibonacci numbers ($Fib(m)$ and $Fib(n)$) are converted to strings and the concatenated into one string. The same computation si made on the server side. After the client's application returns the result, the new generated string is sent to server. Then server compares the client's result and its eigen result. If the both strings are identical then user is granted to access.

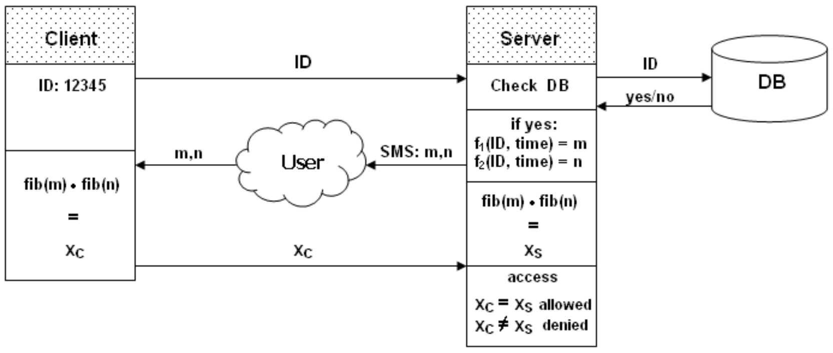

Figure 11: The model of an authentication system

We would like to remark that our model of system is only a proposition of an algorithm and we do not dwell on the data and transfer encryption.

7.2 How the authentication process calculates the user input values

The computation of the values sent to the user is based on mathematical structures introduced in previous chapters. The computation uses corecursive and recursive structures. An anamorphism is a generating structure and catamorphism is an eliminating structure. Firstly, the list of values is being generated by anamorphism and then the list is being consumed - eliminated by catamorphism. The elimination operation of catamorphism in that case is an addition.

For the computation of Fibonacci numbers we use the Pascal's triangle. We showed in [19] that it holds for the i-th Fibonacci number:

1. if the number i is even:
$$fib(i) = \sum_{\substack{k=0 \\ n=i-k-1}}^{\frac{i}{2}-1} \binom{n}{k} \tag{1}$$
2. if the number i is odd:
$$fib(i) = \sum_{\substack{k=0 \\ n=i-k-1}}^{\frac{i-1}{2}} \binom{n}{k} \tag{2}$$

For the combination number calculation we apply the previously defined factorial based on hylomorphysm

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

and in $OCaml$:

```
let comb (n,k) = (fact (n)) / ((fact (n–k)) * (fact (k)) );;
```

The usage of the function $comb(n, k)$ is as usual:

```
# comb (3,2);;
- : int = 3
```

and the signature of the function is $int \times int \rightarrow int$.

The function fib_ana is based on generating structure of anamorphism. This function corresponds to calculation of the n-the Fibonacci number by using formulas (1) and (2); its signature is $int \rightarrow (int \times int)List$. It generates the list of elements, where each element of the list is a tuple representing the combination number. Such a generated list is an input for the function fib_cata. In the function fib_ana are used subsidiary functions $repeat\_up$, $repeat\_down$, and zip.

```
let fib_ana i =
if ((i mod 2) = 0) then
zip (repeat_down (i/2) (i-1)) (repeat_up (i/2) 0)
else
zip (repeat_down (((i-1)/2)+1) (i-1)) (repeat_up (((i-1)/2)+1) 0);;
```

The usage of the function fib_ana is as follows:

```
# fib_ana 6;;
- : (int * int) list = [(5, 0); (4, 1); (3, 2)]
```

and the result is the list of three tuples which represent the combination numbers.

The subsidiary function $repeat\_up$ is a simple recursive function which inserts n elements into the list. The numbers are generated increasingly starting from the value x.

```
let rec repeat_up n x =
  if n > 0
  then x::repeat_up (n-1) (x+1)
  else [];;

# repeat_up 3 0;;
- : int list = [0; 1; 2]
```

Similarly, the subsidiary function $repeat\_down$ inserts n elements into the list. The numbers are generated decreasingly starting from the value x.

```
let rec repeat_down n x =
  if n > 0
  then x::repeat_down (n-1) (x-1)
  else [];;

# repeat_down 3 5;;
- : int list = [5; 4; 3]
```

Finally, the function zip is a recursive function. Its input are two lists: the first is a result of the function $repeat\_up$ and the second is a result of $repeat\_down$. The result of the function zip is a new list which elements are tuples consisting of elements of both input list - the elements of input lists are "zipped" into a tuples and inserted into one list. The function has a signature $intList \times intList \rightarrow (int \times int)List$.

```
let rec zip lst1 lst2 = match lst1,lst2 with
  | [],_ -> []
  | _, []-> []
  | (x::xs),(y::ys) -> (x,y) :: (zip xs ys);;

# zip [5; 4; 3] [0; 1; 2];;
- : (int * int) list = [(5, 0); (4, 1); (3, 2)]
```

The function $cata\_fib$ is a recursive function. Its input is a list of tuples of integers that represent the combination number from the Pascal's triangle. Function is based on the eliminating structure - the catamorphism. Input list is eliminated by calculating the combination number from the concrete tuple by using the function $comb$ and then adding all that values into one sum. After all the list is consumed, the function fib_cata returns the n-th Fibonacci number. Our method does not need to calculate previous two Fibonacci numbers for the n-the Fibonacci numbers. The result is obtained just by adding the concrete combination numbers. The signature of the function fib_cata is $(int \times int)List \rightarrow int$

```
let rec fib_cata list =
  match list with
  | [] -> 0
  | head :: tail -> comb (List.hd list) + (fib_cata tail);;
```

```
        # fib_cata [(5, 0); (4, 1); (3, 2)];;
        - : int = 8
```

Finally, the function fib is a composition of the functions fib_ana and fib_cata. It generates the list of tuples which represent the combination numbers and then the list is eliminated by adding all the values into one sum. The result of the function fib is the n-th combination number without knowing its previous two numbers in the sequence of the Fibonacci numbers. Signature of the function fib is $int \rightarrow int$.

```
let fib i =
  fib_cata (fib_ana i);;
```

```
        # fib 6;;
        - : int = 8
```

8. Conclusion

In this contribution we presented an alternative way of the factorial calculation. Our way is based on the algebraic and coalgebraic structures: the anamorphism, the catamorphism and the hylomorphism which are algebraical and coalgebraical structures and they can be 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 formulas in linear logic. By using the recursive and corecursive structures we proposed simple authentication system. The idea of our authentication system is that user is not requested to input the password but to make some calculations and then send those results to server. If the results have been calculated correctly, the user is granted to access. Our next goal is going to be the extension of this approach in other categorical structures based on recursive coalgebras and constructing of the appropriate description of the computation in linear logic and to describe the behavior of the authentication system by using categorical structures.

Acknowledgments

This work has been supported by VEGA Grant No.1/0015/10: Principles and methods of semantic 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] ALEKSIĆ, S., RISTIĆ, S., AND LUKOVIĆ, I. An approach to generating server implementation of the inverse referential integrity constraints. In *The 5th International Conference on Information Technologies - ICIT 2011* (Amman, Jordan, May 2011).

[3] AWODEY, S. *Category Theory*. Carnegie Mellon University, 2005.

[4] BARR, M., AND WELLS, C. *Category Theory for Computing Science*. Prentice Hall International, 1990. ISBN 0-13-120486-6.

[5] 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.

[6] FOKKINGA, M., AND MEIJER, E. Program calculation properties of continuous algebras. Tech. rep., CWI, Amsterdam, 1991. Technical Report CS-R9104.

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

[8] GIRARD, J., TAYLOR, P., AND LAFONT, Y. *Proofs and Types*. Cambridge University Press, 1990.

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

[10] KABANOV, J., AND VENE, V. Recursion schemes for dynamic programming. In *Proc. of 8th Int. Conf. on Mathematics of Program Construction - MPC'06* (Tartu, Estonia, 2006), T. Uustalu, Ed.

[11] LAMBEK, J., AND SCOTT, P.-J. Introduction to higher-order categorical logic. *Studies in Adv. Math, Cambridge University Press*, No. 7 (1986).

[12] LUKOVIĆ, I., RISTIĆ, S., ALEKSIĆ, S., BANOVIĆ, J., AND POPOVIĆ, A. A chain of model transformations in IIS*case. *Scripta Scientiarum Naturalium 1*, 1 (2010), 59–76. Faculty of Natural Sciences and Mathematics, Podgorica, Montenegro.

[13] MATOUŠEK, J., AND NEŠETŘIL, J. *Kapitoly z diskrétní matematiky*. Nakladatelství Karolinum, Praha, Univerzita Karlova v Praze, 2000.

[14] NOVITZKÁ, V. Church's types in logical reasoning on programming. In *Acta Electrotechnica et Informatica* (2006), Košice, pp. 27–31.

[15] NOVITZKÁ, V., AND SLODIČÁK, V. *Categorical structures and their application in informatics*. Equilibria, Košice, 2010.

[16] NOVITZKÁ, V., SLODIČÁK, V., AND VERBOVÁ, A. On modeling higher-order logic. In *Informatics 2007, Proceedings of the Ninth International Conference on Informatics* (2007), Slovak Society for Applied Cybernetics and Informatics Bratislava, pp. 156–162.

[17] 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.

[18] SÁGI, G. A completness theorem for higher order logic. *J. Symbolic Logic 65*, 2 (2000), 857–884.

[19] 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.

[20] SLODIČÁK, V., AND L'AL'OVÁ, 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.

[21] SØRENSEN, M., AND URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. University of Copenhagen a University of Warsaw, 1999.

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