Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 41–62

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Some New Approaches in Functional Programming Using Algebras and Coalgebras

Viliam Slodičák[1,2] Pavol Macko[3]

Department of Computers and Informatics
Technical University of Košice
Košice, Slovak Republic

Abstract

In our paper we deal with the expressing of recursion and corecursion in functional programming. We discuss about the morphisms which express the recursion or corecursion, respectively. Here we consider especially the catamorphisms, anamorphisms and their composition called the hylomorphisms. The main essence of this work is to describe a new method of programming the function for calculating the factorial by using hylomorphism. We show that using of hylomorphism is an alternative method for the computation of factorial to recursive methods programmed classically. Our new method we describe in action semantics which is a new formal method for the program description.

Keywords: Recursion, duality, hylomorphism, algebra, recursive coalgebras, action semantics.

1 Introduction

Recursion is a very useful tool in modern programming languages, in particular when we deal with inductive data structures such as lists, trees, etc. [9]. The theory of recursive functions was developed by Kurt Gödel and Stephen Kleene in the 1930s. Now the recursion has the great importance in mathematics and in computer science. The recursion method presents very interesting solution of many computational algorithms. It is an inherent part of functional programming. On the other hand, the dual method called *corecursion* brings the new opportunities in programming. In section 2 we formulate basic notions from the category theory, the mathematical tool which importance for theoretical informatics growth in last decade. Section 3 describes the relationship between recursion and corecursion. This relationship is described by the dual mathematical structures - algebras and

[1] 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.

[2] Email: viliam.slodicak@tuke.sk

[3] Email: pavol.macko@tuke.sk

doi:10.1016/j.entcs.2011.11.037

coalgebras [32]. Algebras serve for modeling program structures and coalgebras have great addition for defining the behavioral semantics [21]. Coalgebras are able to describe the behavior of simple structures like automata [1], intrusion detection system [20], computer simulations [13] up to database systems [19]. By using coalgebras we usually model structures which cannot be modeled by algebras; e.g. infinite data structures (*codata*) as the base of corecursion [16,26,29]. It is important to note that data types and codata types are modeled as initial algebras and terminal coalgebras, respectively. On the other hand, recursion and corecursion are modeled by morphisms called catamorphisms and anamorphisms, resp. In the next section we construct a recursive coalgebra based on the unique coalgebra-to-algebra morphisms called *hylomorphism*. In the last section we show the examples how to use introduced morphisms in functional programming and create a new non-traditional method of programming using recursion and corecursion. Computation and evaluation of results we present in action semantics which is a new method for defining the semantics of programs. Action semantics is a framework for the formal description of programming languages. Its main advantage over other frameworks is pragmatic: action-semantic descriptions are able to scale up smoothly to realistic programming languages. In our paper we show the evaluation of our alternative method for the factorial computation.

2 Categories

Algebraic and coalgebraic concepts in informatics are based on the category theory [4,14]. A category $\mathscr{C}$ is a mathematical structure consisting of objects, e.g. $A, B, C, \ldots$ Instead of focusing merely on the individual objects representing a given structure, category theory emphasizes the morphisms - the structure-preserving mapping between objects, $f : A \to B$ [5]. Every object has the identity morphism $id_A : A \to A$ and morphisms are composable - for morphisms $f : A \to B$ and $g : B \to C$ it has to exist $f \circ g : A \to C$. Composition of morphisms has the associativity property. 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. Because a category itself is a mathematical structure we are able to construct the morphisms between categories called functors, e.g. a functor $F : \mathscr{C} \to \mathscr{D}$ from a category $\mathscr{C}$ into a category $\mathscr{D}$.

3 Recursion versus corecursion

Recursive functions are functions for which the result for a certain argument depends on the result obtained for other (smaller in some sense) arguments. Recursion in mathematics and theoretical computer science is a method of defining functions which the function being defined in is applied within its own definition. It is very important concept in applications ranging from the theoretical foundations of computation to practical programming techniques. Since last years, it has become increasingly clear that the dual but less well-known concept of corecursion is just as useful. Moss and Danner used in their work [22] the concept of *corecursive program* for a function whose range is a type defined recursively as the greatest solution of some equation which is expressed as the greatest fixed point. Dually, we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation (least fixed point) [12]. To state this in a different manner, corecursion produces the (possibly) infinite data structures and recursion consumes those data structures.

3.1 Initial algebras and catamorphisms

Let F be an endofunctor from $\mathscr{C}$ to $\mathscr{C}$. An F-algebra is a pair (A, α) where A called the carrier is an object and the algebraic 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 a homomorphism between F-algebras - from (A, α) to (C, γ), if the following diagram at Fig. 1 commutes.

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

Fig. 1. Diagram of algebras

The commutativity of the diagram at Fig. 1 means that the following equality holds

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

Because algebras are homogenous structures and the algebra homomorphisms are being defined, we are able to define also an identity morphism for each algebra and appropriate morphisms are composable. That implies, that we can construct the category of F-algebras $\mathscr{A}lg(F)$, where algebras are the category objects and algebra homomorphisms are the category morphisms.

An F-algebra is said to be the *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, if they exist, must satisfy the following noteworthy properties:

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

Put differently, from the first property it follows that there exists at most one initial F-algebra. Because from the initial F-algebra there exists unique homomorphism to every F-algebra, the initial F-algebra is the initial object in the category

$\mathscr{A}lg(F)$. It was proved in [17] 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. Let in_F be the algebraic structure of initial algebra. The least fixed point of the functor F we denote μF. Given any endofunctor $F : \mathscr{C} \to \mathscr{C}$ on an arbitrary category $\mathscr{C}$, if $(\mu F, in_F)$ is an initial F-algebra, then $in_F : F\mu F \to \mu F$ is an isomorphism, $F\mu F \cong \mu F$ [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 [10], we denote this homomorphism by $(cata\ \alpha)_F$ (or just $cata\ \alpha$ if the functor F is clear), so $(cata\ \alpha)_F : \mu F \to A$ is being characterized by the universal property

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

The type information is illustrated in the commutative diagram at Fig. 2. In [31] are proved properties of catamorphism.

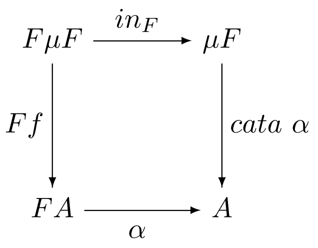
$$\begin{tikzcd}
F\mu F \arrow[r, "in_F"] \arrow[d, "Ff"'] & \mu F \arrow[d, "cata\ \alpha"] \\
FA \arrow[r, "\alpha"'] & A
\end{tikzcd}$$

Fig. 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 Terminal coalgebras and anamorphisms

Coalgebras are dual structures to algebras. Let F be an endofunctor from $\mathscr{C}$ to $\mathscr{C}$. An F-coalgebra is a pair (U, φ) where U is called the state space and $\varphi : U \to FU$ is called the coalgebra structure (or *coalgebra dynamics*). For any two F-coalgebras (U, φ) and (V, ψ), a morphism $g : U \to V$ is said to be a homomorphism from (U, φ) to (V, ψ) between F-coalgebras, so the diagram at Fig. 3 commutes

so it holds the equality

$$\varphi \circ Fg = g \circ \psi.$$

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 the definition of category. An F-coalgebra is said to be a terminal

$$\begin{tikzcd} U \arrow[r, "\varphi"] \arrow[d, "g"'] & FU \arrow[d, "Fg"] \\ V \arrow[r, "\psi"'] & FV \end{tikzcd}$$

Fig. 3. Diagram of coalgebras

F-coalgebra if it is the terminal object of the category $\mathscr{C}oalg(F)$. The existence of the terminal 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 denoted by $(ana\ \varphi)_F$. By analogy, the morphism $(ana\ \varphi)_F : U \to \nu F$ is characterized by the universal property

$$g \circ out_F = \varphi \circ Fg \quad \Leftrightarrow \quad g = (ana\ \varphi)_F.$$

The type information is illustrated in the commutative diagram at Fig. 4.

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

Fig. 4. Diagram of terminal coalgebra and anamorphism

Morphisms of the form $(ana\ \varphi)_F$ (or just $ana\ \alpha$ if the functor F is clear) 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 the same category. In particular, in functional programming one often uses the universal property of an initial algebra to provide a semantics of a recursive program. Recursive coalgebras extend that universal property beyond the initial algebra considered as coalgebra [2,6].

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

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

Fig. 5. Diagram of recursive coalgebra

For recursive coalgebra it holds the equality

$$r = \varphi \circ Fr \circ \alpha.$$

A unique coalgebra-to-algebra morphism from (U, φ) to (A, α) can be denoted also as $fix_{F,\alpha}(\varphi)$ [6].

3.4 Hylomorphism

The hylomorphism recursion pattern was firstly defined in [11]. Given a functor F, it expresses the following recursive function of type $U \to A$ defined by fixed point [8]

$$hylo(g, h)_F = \mu(\lambda f.h \circ Ff \circ g).$$

for the function $g : FA \to A$ and for the function $h : U \to FU$.

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 $U \to A$ that makes the following diagram at Fig. 6 commutes [15].

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

Fig. 6. Diagram of hylomorphism

Moreover, the hylomorphism is a composition of an anamorphism with a catamorphism [15]:

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

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

4 The hylomorphism in functional programming

In this section we apply the theory described in previous sections and we present them in one example. We use the hylomorphism as a composition of anamorphism and catamorphism, that is exactly the same as recursive coalgebras, which are based on the coalgebra morphisms called coalgebra-to-algebra morphism. Coalgebra is expressed as an anamorphism and algebra as a catamorphism.

The common feature of all functional programming languages is that programs consist of functions (as in the mathematical notion function, which is the basis of the λ-calculus and the theory of partial recursive functions; not to be confused with the notion of a function used in imperative languages). The most modern functional programming languages are strongly typed and have built-in memory management; for example the languages ML, $Haskell$ and $OCaml$ [9]. We show on an example the fancy method of programming the function for calculating the factorial using hylomorphism. Hylomorphism in program calls the function that consists of a composition of two functions. Therefore it has to create the first function which uses anamorphism and then second function which uses catamorphism. Thus combining these two functions we get a function that expresses the hylomorphism in program.

We present in this work all code samples written in the *OCaml* language [18]. The *OCaml* language (objective *Caml*) is an object-oriented functional language based on the *Caml* language.

4.1 The anamorphism in example

Anamorphism is a constructor based on operation as it was defined in the description of anamorphism. By applying the anamorphism in the informatics we get a corecursive function that starts with a single input (for example an integer) and takes it in a much complex output, for example a wide list. In other words, we take a single input and we expand it upwards into a wide list. We define this process in functional programming as the $unfold$ function. It takes a type signature of $< TypeB >$ into $List < TypeA >$. In the *OCaml* language the type signature for an anamorphism is $a \to b\ list$.
The application of an anamorphism is the function for generating natural numbers. For illustration we named this function `ana` in the implementation, because the basis of this function is the anamorphism.
Argument of this function is an element of the type int. The function returns the list of elements from n to 1 as an output. Such a generated list we need for the calculation of the factorial of n. The concrete type signature of this function is:

$$int \to intList.$$

So it is a function that takes an integer and produces a list of integers. The implementation of the function `ana` in the *OCaml* language is:

```
let rec ana n =
match n with
```

```
| 0 -> []
| 1 -> [1]
| x -> x :: ana (x-1);;
```

If the argument of the function `ana` is 0 then it returns an empty list. If the argument is 1, `ana` generates a list containing only 1 as item. Otherwise, `ana` generates a list with a new element appended.

4.2 The catamorphism in example

Catamorphism is deconstructor based on operation as it was defined in the description of catamorphism.

By applying the catamorphism in the informatics, we get a recursive function that starts with a list and returns it in a single output (for instance integer). By applying the catamorphism on the list of elements of the type *int* we get the single element of type *int*. The *max*() or *length*() methods in *OCaml* on a list are catamorphisms. These functions are called *fold* in functional programming. They take a type signature of $List < TypeA >$ into $< TypeB >$. In the *OCaml* language, the type signature for the catamorphism is $a\ list \to b$. We present catamorphism as the multiplication function. This function takes as its argument a list of factors of the type *int* and returns the result of multiplicative operations. The result of the function is an element of the type *int*, that is the result of multiplication of elements in the list. The concrete type signature of this function is then

$$intList \to int.$$

So it takes a list of integers and folds it into a single integer. For better illustration we named the multiplication function as `cata` in the implementation, because the basis of this function is catamorphism:

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

For the purposes of this example the `cata` function takes a list and returns 1 if the list is empty. Otherwise, it recursively calls the `cata` on the list.

4.3 The hylomorphism in example

After presenting the catamorphism and anamorphism by the examples we define hylomorphism. The hylomorphism is described like as '*cata*' after '*ana*'. So putting an anamorphism ($a \to c\ list$) together with a catamorphism ($c\ list \to b$) they result into a type signature of $a \to b$. Composition of $unfold$ with a $fold$ gives a function that takes a single value and returns also the single value; the recursive function that builds and then reduces lists is hidden for user.

We use previous implementations to define the hylomorphism. The function `ana` generates list of natural numbers from n to 1 and the function `cata` eliminates that generated list by using multiplication operation. The result of this hylomorphism function is the factorial of a number n. Composition of two function $f \circ g$ is written in programming language *OCaml* as $f(gx)$.
The definition of this hylomorphism function $fact$ is as follows:

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

Execution of the function of factorial:

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

Execution of the previous example with hylomorphism:

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

Her we are able to compare our new approach with the classical approach without hylomorphism. The implementation of factorial function without hylomorphism:

```
let rec factorial x =
if x = 1 then 1
else x * (factorial (x-1));;
```

Illustration of this example without hylomorphism:

```
factorial 4 =
   4 * (factorial 3) =
   4 * 3 * (factorial 2) =
   4 * 3 * 2 * (factorial 1) =
   4 * 3 * 2 * 1 =
   4 * 3 * 2 =
   4 * 6 =
   24
```

We can see that our new method of the factorial computation by using the hylomorphism gives the expected results.

5 Basic concepts about Action semantics

The framework of action semantics has been initially developed at the University of Aarhus by Peter D. Mosses, in collaboration with David Watt from University of Glasgow. An action semantics is a framework for the formal description of programming languages. Its main advantage over other frameworks is pragmatic: action-semantic descriptions (ASDs) can scale up easy to real programming languages [7,23,27,30]. This is due to the inherent extensibility and modifiability of ASDs, ensuring that extensions and changes to the described language require only proportionate changes in its description. On the other hand, adding an unforeseen construct to a language may require a reformulation of the entire description in denotational or operational semantics expressed in [24,25].

Action semantics is fully equivalent with other semantic methods, like denotational semantics, operational semantics or axiomatic semantics. Fundamentals of action semantics are actions which are essentially dynamic computational entities. They incorporate the performance of computational behavior, using values passed to them to generate new values that reflect changes in the state of the computation. So the performance of an action directly represents the information of processing the behavior and reflects the gradual, step-wise nature of computation: each step of an action performance may access and/or change the current information. Other semantic entities used in action semantics are yielders and data. The actions are main kind of entities, the yielders and data are subsidiary. The notation used for specifying actions and the subsidiary semantic entities is called action notation [23]. In action semantics, the semantics of a programming language is defined by mapping program phrases to actions. The performance of these actions relates closely to the execution of the program phrases. Primitive actions can store data in storage cells, bind identifiers to data, compute values, test truth values, and so on [28].

The data entities consist of mathematical values, such as integers, Boolean values, and abstract cells representing memory locations, that embody particles of information. Sorts of data used by action semantics are defined by algebraic specifications. Yielders encompass unevaluated pieces of data whose values depend on the current information incorporating the state of the computation. Yielders are occurring in actions and may access, but they are not allowed to change the current information.

A performance of an action which may be part of an enclosing action either:

- *completes*, corresponding to normal termination;
- *escapes*, corresponding to exceptional termination;
- *fails*, corresponding to abandoning an alternative;
- *diverges*, corresponding to deadlock.

The different kinds of information give rise to so-called *facets* of actions which have been classified according to [23]. They are focusing on the processing of at most one kind of information at a time:

- *the basic facet*, processing independently of information (control flows);
- *the functional facet*, processing transient information (actions are given and give data);
- *the declarative facet*, processing scoped information (actions receive and produce

bindings);

- *the imperative facet*, processing stable information (actions reserve and dispose cells of storage, and change the data stored in cells);
- *the communicative facet*, processing permanent information (actions send messages, receive messages in buffers, and offer contracts to agents) [23].

The standard notation for specifying actions consists of primitive actions and action combinators. Action combinators combine existing actions, normally using infix notation, to control the order which subactions are performed in as well as the data flow to and from their subactions. Action combinators are used to define sequential, selective, iterative, and block structuring control flow as well as to manage the flow of information between actions. The standard symbols used in action notation are ordinary English words. In fact, action notation is very near to natural language:

- terms standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g. `check it and then escape`;
- terms standing for data and yielders form noun phrases, e.g. `the items of the given list`.

These simple principles for choice of symbols provide a surprisingly grammatical fragment of English, allowing specifications of actions to be made fluently readable. The informal appearance and suggestive words of action notation should encourage programmers to read it. Compared to other formalisms, such as λ-notation, action notation may appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also take into account that each action combinator usually corresponds to a complex pattern of applications and abstractions in λ-notation. In any case, the increased length of each symbol seems to be far outweighed by its increased perspicuity.

6 Action semantics in functional paradigm

Action semantics can be successfully used also for the description of functional programs. In action semantics we use generally three main actions for the description of programming languages:

- *execute* - used for executing of statements;
- *elaborate* - used with declarations;
- *evaluate* - used for evaluating expressions.

In functional paradigm we use only two main actions: *evaluate* and *elaborate*. Action *execute* is not important in functional paradigm. Typical for functional programs is that they do not deal with storage. Therefore we will not use actions of imperative facet for allocating memory locations, storing values and getting values from cells in memory in our action semantics descriptions of functional programs.

Important for functional paradigm is an evaluating of the expressions and elaborating functions. To allow referring them in the program code, they are associated to names (identifiers). These associations are called bindings. A binding can be *global*, when declared at the top level of the source code, or *local*, when declared in a *let* or *letrec* expressions that contain it. The difference between *let* and *letrec* expressions is that in the latter mutual recursion is allowed. We provide this description of evaluation of simple expression:

elaborate⟦`let` I`:Var = E:Expression`⟧ =

  evaluate ⟦ `E` ⟧

    `then bind` I `to the given value`

After declaration we are able to use it anytime in our program. The value is bound to its identifier, so we can get the value of this expression simply by using *evaluate* action:

evaluate⟦ I`:Var` ⟧ =

  `give the value bound to` I

Description of function with one argument should seem like this:

elaborate⟦`let` I_f`:Var` I_{p1}`:Var = E:Expression`⟧ =

  evaluate⟦`E`⟧

    `then bind` I_f `to the given value`

In the expression E is used parameter of the function which value we can get simply with action *evaluate*:

evaluate⟦ I_{p1}`:Var` ⟧ =

  `give the value bound to` I_{p1}

General definition for function with different number of arguments:

elaborate⟦`let` I_f`:Var` $< I_p$`:Var` $>^+$ `= E:Expression`⟧ =

  evaluate⟦`E`⟧

    `then`

  `bind` I_f `to the given value`

6.1 The description in Action semantics

Let E_1 be the substitution for the function *ana*:

$E_1 =$ `match` n `with`

  | 0 -> []

  | 1 -> [1]

  | x -> x :: *ana* (x-1)

We elaborate the function declaration in Action semantics as

```
elaborate ⟦ let rec $ana$ $n$ = $E_1$⟧ =
  recursively bind $ana$ to
    closure of
      abstraction of
        evaluate⟦$E_1$⟧ =
  recursively bind $ana$ to
    closure of
      abstraction of
        evaluate⟦ match $n$ with | $0 -> []$ | $1 -> [1]$ |$x ->$ $x$ :: $ana(x-1)$ ⟧
```

and the evaluation of the action **evaluate**⟦E_1⟧ is:

```
evaluate⟦ match $n$ with | $0 -> []$ | $1 -> [1]$ | $x ->$ $x$ :: $ana(x-1)$ ⟧ =
evaluate ⟦$n$⟧
  and then
  (check the given value is equal to the number 0
    and then give the empty list
      or
  check the given value is equal to the number 1
    and then add the number 1 to the list)
      or
  check the given value is greater than the number 1
    and then add the given number to the list
      before
    add evaluate ⟦$ana$ $(n-1)$⟧ to the list
```

The description of function *cata* in Action semantics is analogous. First we define a substitution E_2 for the *cata* function as follows:

```
Let $E_2$ =   match myList with
              | [] -> 1
              | head ::  tail -> head * ($cata$ tail)
```

We also define primitive actions `head_` and `tail_` for the treatment the data structure list:

```
head list =
    give the first element of the list
```

which gives the first element of the given list, and

```
tail list =
    remove first element from the list   which gives the tail of the list,
      then give the list
```

i.e. all elements except the first one are being returned. Now we are able to elaborate the declaration of the function *cata* and we obtain full description of it in Action semantics:

```
elaborate ⟦ let rec $cata$ $myList$ = $E_2$⟧ =
  recursively bind $cata$ to
    closure of
      abstraction of
        evaluate⟦$E_2$⟧
```

The evaluation of the action $[\![E_2]\!]$ is defined as follows:

```
evaluate⟦$E_2$⟧ =
  give the value bound to $myList$
    and then
  give the $TruthValue$ of (the given list is empty)
    then
  check the given $TruthValue$
      and then give the number 1
    or
  check not the given $TruthValue$
      and then
        give the multiplication of
        (head the given list
          and
        evaluate⟦ $cata$ (tail the given list)⟧)
```

Finally, we define the function $fact$ for the computation of the factorial. The elaboration of the function $fact$ declaration is:

```
elaborate ⟦ let $fact\ x = cata\ (ana\ x)$⟧ =
 evaluate ⟦$cata\ (ana\ x)$⟧
   then
  bind $fact\ x$ to the given value =
 evaluate ⟦$ana\ x$⟧
   before
 evaluate ⟦$cata$ (the given list)⟧
     then
       bind $fact\ x$ to the given value
```

where the actions **evaluate**⟦*ana x*⟧ and **evaluate**⟦*cata myList*⟧ be evaluated in the following way:

```
evaluate ⟦$ana\ x$⟧
  give the value bound to
    closure of
      abstraction of
        evaluate⟦$E_1$⟧
evaluate ⟦$cata\ myList$⟧
  give the value bound to
    closure of
      abstraction of
        evaluate⟦$E_2$⟧
```

After defining all actions necessary for the description of the factorial computation we present an example for the factorial of given input value.

6.2 Example in Action semantics

In this section we present the evaluation of factorial for the input value $n = 4$. Our alternative computation method of the factorial has been defined in chapter 4.

```
evaluate ⟦ $fact\ x = cata\ (ana\ x)$⟧$s\,[x \mapsto 4]$ =
  give the value bound to
    evaluate ⟦$cata\ (ana\ x)$⟧ =
give the value bound to
   ($\mathbf{evaluate}$ $[\![ana\ x]\!]s\,[x \mapsto 4]$
     before
   $\mathbf{evaluate}$ $[\![cata\ \textit{(the given list)}]\!]$)
```

where $\mathbf{evaluate}[\![e]\!]s[variable \mapsto value]$ means the evaluation of an expression e in the input state where the given *variable* is set to the given *value*. Evaluation of the function $ana\ x$ for the input value $x = 4$ is:

```
$\mathbf{evaluate}[\![ana\ x]\!]s\,[x \mapsto 4] =$
give the value bound to
   closure of
     abstraction of
       $\mathbf{evaluate}[\![$ match $x$ with $|\ 0$ -> $[]\ |\ 1$ -> $[1]\ |x$ -> $x :: ana(x-1)\ ]\!]s\,[x \mapsto 4] =$
give the value bound to
  closure of
    abstraction of
      give the value bound to $x\ s\,[x \mapsto 4]$
    and then
      check the given number is greater than the number $1$
        and then add the given number to the list
      before
        add $\mathbf{evaluate}\ [\![ana\ (x-1)]\!]s\,[x \mapsto 4, list \mapsto [4]]$ to the list $=$
give the value bound to
  closure of
    abstraction of
      give the value bound to $x\ s\,[x \mapsto 3]$
    and then
      check the given number is greater than the number $1$
        and then add the given number to the list
      before
        add $\mathbf{evaluate}\ [\![ana\ (x-1)]\!]s\,[x \mapsto 3, list \mapsto [4,3]]$ to the list $=$
```

```
give the value bound to
 closure of
  abstraction of
   give the value bound to $x$ $s\,[x \mapsto 2]$
  and then
   check the given number is greater than the number $1$
    and then add the given number to the list
   before
    add **evaluate** $[\![ana\ (x-1)]\!]s\,[x \mapsto 2, list \mapsto [4,3,2]]$ to the list $=$
give the value bound to
 closure of
  abstraction of
   give the value bound to $x$ $s\,[x \mapsto 1]$
  and then
   check the given number is equal to the number $1$
    and then add the given number to the list $=$
give the value bound to
 closure of
  abstraction of
   give the list $[4,\ 3,\ 2,\ 1]$
```

The final state is $s\,[x \mapsto 1, list \mapsto [4,3,2,1]]$.

Next step is the evaluation of action **evaluate**$[\![cata\ list]\!][list \mapsto [1,2,3,4]]$. In its evaluation we apply the primitive actions `head_` and `tail_`, which have been defined in chapter 6.1.

In next step, the action **evaluate** $[\![cata\ list]\!]s\,[list \mapsto [4,3,2,1]]$ is being evaluated. The actions `head_` and `tail_` which have been defined in chapter 6.1 are being used.

```
evaluate ⟦$cata\ list$⟧$s\,[list \mapsto [4,3,2,1]]$
 give the value bound to
   closure of
     abstraction of
       evaluate⟦ match list with | [] ->
         1 | head::tail -> head * ($cata$ tail) ⟧=
 give the value bound to
   closure of
     abstraction of
       give the $TruthValue$ of (list is empty)
         then
       check not the given $TruthValue$
         and then
           give the multiplication of
           (head list $s\,[list \mapsto [4,3,2,1]]$
             and
           evaluate ⟦$cata$ (tail list)⟧$s\,[list \mapsto [4,3,2,1]]$) =
 give the value bound to
   closure of
     abstraction of
       give the $TruthValue$ of (list is empty)
         then
       check not the given $TruthValue$
         and then
           give the multiplication of
           (the number $4$
             and
```

```
          head list $s\,[list \mapsto [3,2,1]]$
            and
          evaluate $[\![cata$ (tail list)$]\!]s\,[list \mapsto [3,2,1]]) =$
give the value bound to
  closure of
    abstraction of
      give the $TruthValue$ of (list is empty)
        then
      check not the given $TruthValue$
        and then
          give the multiplication of
          (the number $4$
            and
          the number $3$
            and
          head list $s\,[list \mapsto [2,1]]$
            and
          evaluate $[\![cata$ (tail list)$]\!]s\,[list \mapsto [2,1]]) =$
give the value bound to
  closure of
    abstraction of
      give the $TruthValue$ of (list is empty)
        then
      check not the given $TruthValue$
        and then
          give the multiplication of
          (the number $4$
            and the number $3$
```

```
              and the number $2$
              and head list $s\,[list \mapsto [1]]$
              and
             $\textbf{evaluate}[\![$ $cata$ (tail list)$]\!]s\,[list \mapsto [1]]) =$
   give the value bound to
     closure of
       abstraction of
         give the $TruthValue$ of (list is empty)
           then
         check the given $TruthValue$
           and then
             give the multiplication of
             (the number $4$
               and the number $3$
               and the number $2$
               and the number $1$
               and the number $1) =$
   give the value bound to
     closure of
       abstraction of
         give the number $24$
```

We can see that the description in action semantics seems to be very long, it is very good readable for the programmers and the results obtained by this method are coorect and correspond to real computations.

7 Conclusion

In this paper we have focused on the analysis of recursion and corecursion. We described the recursion by initial algebras and catamorphisms. Then we described the corecursion as a dual method of recursion by terminal coalgebras and anamorphisms. To define the relationship between recursion and corecursion we used algebras and coalgebras which are dual structures. The exact relation between algebra and coalgebra we defined by the recursive coalgebras which is based on unique coalgebra-to-algebra morphism. This morphism is otherwise known as hylomorphism. In the last chapters we showed an unusual example for calculating the factorial of number n with our new method using anamorphism, catamorphism and hylomorphism; the description of this method we presented in action semantics. We presented an alternative method of how to make a computation of recursive functions by special mathematical structures - the algebras and coalgebras with the relation between them expressed by recursive coalgebras. Our future research will be the exact categorical formulation of those principles by using the structures for the contrustion of the algebras and coalgebras: monads and comonads.

Acknowledgment

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., *Introduction to coalgebra*, Theory and Applications of Categories **Vol. 14** (2005), pp. pp. 157–199.

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

[3] Awodey, S., "Category Theory," Carnegie Mellon University, 2005.

[4] Barr, M., "Terminal coalgebras for endofunctors on sets," McGill University, Montreal, Quebec, Canada (1999).

[5] Barr, M. and C. Wells, "Category Theory for Computing Science," Prentice Hall International, 1990, iSBN 0-13-120486-6.

[6] Capretta, V., T. Uustalu and V. Vene, *Recursive coalgebras from comonads*, Inf. Comput. **204** (2006), pp. pp. 437–468.

[7] Cheng, F., *Mda implementation based on patterns and action semantics*, , **2**, 2010, pp. 25–28.

[8] Cunha, A., "Point-free Programming with Hylomorphisms," Ph.D. thesis, Universidade do Minho, Braga, Portugal (2005).

[9] Fernandez, M., "Models of Computation. An Introduction to Computability Theory." Springer-Verlag London Limited, 2009, iSBN 978-1-84882-433-1.

[10] Fokkinga, M., "Law and Order in Algorithmics," Ph.D. thesis, University of Twente, Enschede (1992).

[11] Fokkinga, M. and E. Meijer, *Program calculation properties of continuous algebras* (1991).

[12] Gibbons, J. and G. Hutton, *Proof methods for structured corecursive programs* (1999), oxford University.

[13] Grzybowski, A., *Computer simulations in constructing a coefficient of uncertainty in regression estimation - methodology and results*, in: *Lectures Notes in Computer Science*, 2338 (2002), pp. 671–678.

[14] Jacobs, B. and J. Rutten, *A tutorial on (co)algebras and (co)induction*, Bulletin of the European Association for Theoretical Computer Science (1997), pp. pp. 222–259.

[15] Kabanov, J. and V. Vene, *Recursion schemes for dynamic programming*, in: *Mathematics of Program Construction, 8th International Conference, MPC 2006* (2006), pp. 235–252.

[16] Koubková, A. and J. Pavelka, "Introduction to theoretical informatics," MatFyzPress, Charles University, Prague, 2005, (in Czech).

[17] Lambek, J. and P.-J. Scott, *Introduction to higher-order categorical logic*, Studies in Adv. Math, Cambridge University Press (1986).

[18] Leroy, X., *The objective caml system release 3.12. documentation and users manual.*, Technical report, Institut National de Recherche en Informatique et en Automatique (2008).

[19] Luković, I., P. Mogin, J. Pavićević and S. Ristić, *An approach to developing complex database schemas using form types*, Software: Practise Expererience **Vol. 37** (2007), pp. pp. 1621–1656.

[20] Mihályi, D. and V. Novitzká, *A coalgebra as an intrusion detection system*, Acta Polytechnica Hungarica **Vol. 7** (2010), pp. pp. 71–79, iSSN 1785-8860.

[21] Mihályi, D. and V. Novitzká, "The principles of the duality among the construction and the behavior of the programs," Equilibria, Košice, 2010, (in Slovak).

[22] Moss, L. S. and N. Danner, *On the foundations of corecursion*, Logic Journal of the IGPL **Vol. 5** (1997), pp. pp. 231–257.

[23] Mosses, P. D., *Theory and practice of action semantics*, in: *In MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science* (1996), pp. 37–61.

[24] Nielson, H. R. and F. Nielson, "Semantics with Applications: A Formal Introduction," John Wiley & Sons, Inc., 2003.

[25] Novitzká, V., "Semantics of programs," elfa, Košice, 2001, (in Slovak).

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

[27] Planas, E., J. Cabot and C. Gómez, *Verifying action semantics specifications in uml behavioral models*, in: *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, CAiSE '09 (2009), pp. 125–140.

[28] Ruei, R. and K. Slonneger, *Semantic prototyping: Implementing action semantics in standard ML*, The University of Iowa (1993).

[29] Slodičák, V. and P. Macko, *New approaches in functional programming using algebras and coalgebras*, in: *European Joint Conferrences on Theory and Practise of Software - ETAPS 2011*, Workshop on Generative Technologies, Universität des Saarlandes, Saarbrücken, Germany, 2011, pp. pp. 13–23, iSBN 978-963-284-188-5.

[30] Stuurman, G., *Action semantics applied to model driven engineering* (2010).

[31] Uustalu, T. and V. Vene, *Primitive (co)recursion and course-of-values (co)iteration*, Informatica (1999).

[32] Wisbauer, R., *Algebras versus coalgebras*, Applied Categorical Structures, Springer Netherlands **Vol. 1** (2008), pp. pp. 255–295, iSSN 0927-2852.