Constructive Ontologies

Anatolii Beltiukov
Computer Science Department
Udmurt State University
Izhevsk, Russia
belt.udsu@mail.ru

***Abstract*—An approach to the description of subject areas is proposed to solve constructive tasks in the form of constructive ontologies at the abstraction level of Cartesian closed categories. Constructiveness of ontology makes it possible correct automatic or automated (man-machine) construction of problem solving. In theory the ontology is constructed in such a way that it decides by itself tasks.**

The proposed paradigm is based on the paradigm of deductive synthesis of algorithms. The difference is that the synthesized algorithms are obtained as morphisms of the Cartesian closed category. In essence, ontology is designed to describe the subject area as a Cartesian closed category. In this case, initially objects of a category become what is traditionally thought of as classes of objects of a subject area. Elements of most of these classes themselves are often not considered at the initial stage.

For constructiveness, the ontology is equipped with a logical machine for constructing output. This machine is conceived as an integral part of the most constructive ontology. From a practical point of view, it is important for reliability to have independent logical calculus and a derivation machine. These mechanisms control each other. The result of the logical machine is checked by calculus.

Keywords—ontology, deductive synthesis of algorithms, category theory, Cartesian closure, constructive proof, automatic inference.

I. INTRODUCTION

In this paper, we consider the approach to solving computer science problems within the Cartesian closed categories. Ideas of using Cartesian closed categories for programming have been considered for a relatively long time. These ideas are closely based on the Curry-Howard isomorphism (see [1, 2]), develop in the works [3, 4] and are closely related to the lambda calculus with types (see [5, 6]). To implement these ideas, the concept of a categorical abstract machine is proposed [7].

In the category-theoretic approach as domain domains considered abstract objects, structure which is not originally specified. From point of view programming such an object category is not considered as a set of possible values in the set-theoretic approach, but as an interface between programs, allowing to carry out their composition. Relationship with the set-theoretic model is that the interfaces mentioned are used to transmit data encoding the elements of the sets corresponding to the objects.

Available and created programs in this language are morphisms, connecting some objects. Action of morphism is not limited to mapping values from source object to final one as in functional programming. It can be a complex interaction with transmission of signals, data and programs in opposite directions. For example, a program may make several attempts to produce a result, receiving error signals applying the results previous attempts.

The constructive ontology describes the following entities:

- basic domain objects, in other approaches, these objects correspond domains, data types, etc.,
- parameterized families of objects corresponding to realizations of constructive properties and relationship, with a logical approach, this corresponds to predicates, properties, attributes, relationships between objects,
- basic morphisms corresponding to the program modules already available for solving problems in the domain,
- mechanisms for building new morphisms from available ones,
- an inference machine that determines the order of application of these mechanisms for solving constructive tasks; in difficult cases, an inference machine may have a physico-anthropic-technical nature, that is, to be a system of interacting technical equipment, people and natural objects; on practice we separate its material part and system of programs functioning, with which we will usually identify the logical inference machine.

The category is assumed to be closed under direct product, direct sum, and exponentiation. Exponentiation is the construction of an object corresponding to morphisms between two already existing objects. In addition, closure is required relatively less traditional operations. These are operations of parameterized direct summation and multiplication. These operations are added to be able to formulate and accurately solve problems that can be formulated using logic predicate language.

To use the constructive ontology it is required to specify objects and base morphisms. The inference machine output is automatically translated into common programming languages. For reliable operation the system specying objects and morphisms requires a mechanism verifying compliance of these modules and interfaces. This way of action corresponds to the paradigm of programming without a lower level with abstract types of processed values.

The considered approach is proposed to be used for standardization of automatic and automated solving constructive tasks.

II. BASIC NOTIONS

Initially, we assume that ontology consists of objects, corresponding to the basic concepts. In the set-theoretic approach the objects are usually modeled by some abstract sets. This may correspond to database domains. For example, the object “product” corresponds to some set of products. In the information system, the elements of this set can match information codes of real products from the physical domain.

From our position here the object is primarily informational interfaces allowing connecting various program modules. For example, it may be a data format for transmission messages about a particular product. Filling an object as an interface by pecific values is another task for the next time. The set of these values depends from the specific application of the information system.

So we are implementing a category-theoretic approach here to build an information system. Morphisms under construction category correspond to the program and information modules that carry out transitions from one object to another. To make the information system could function as usual, we require the category being built has some properties of closedness. First of all, it is closure under the direct Cartesian product.

If the object is considered as an interface, in the simplest case Cartesian product corresponds to the parallel connection of two interfaces. In this case, the original data formats can be connected with the advent of the complex record format. This process corresponds to realization of conjunction in constructive mathematics [8]. Therefore the Cartesian product will be denoted by the conjunction sign "&". Thus, if we have objects *A* and *B*, then we have the possibility constructing the object of their direct product:

$$A\&B.$$

Another operation is the direct sum of objects. It corresponds to alternative connection of two interfaces (record with options), in which the interaction is carried out on one or the other interface depending on one of two values of special discriminant element. The direct sum will be denoted by the plus sign "+". Direct sum corresponds to logical disjunction in constructive mathematics [8]. If we have objects *A* and *B*, then we have the opportunity constructing an object of their direct sum:

$$A+B.$$

The next operation is building an exponential, an object, representing morphisms between two source objects. The exponential in our interpretation corresponds to the interface to the module of corresponding morphism. In terms of data transfer it consists of two interfaces acting in the counter directions. In the simple case, the interface of the initial object receives the data and the final object interface provides the data. In constructive logic exponential corresponds to the realization of implication. For the operation of building an exponential we will use the sign "=>". If we have objects *A* and *B*, then we can construct the exponential object

$$A=>B.$$

The statement

$$f:(A=>B)$$

means that *f* is a morphism from object *A* to object *B*. In the general case, $x:A$ means that the object *A* corresponds to some exponential, and *x* can be interpreted as a morphism corresponding to this exponential. On practice $x:A$ is interpreted as stating that object *A* is the problem, and the morphism *x* is its solution. If the task is exponential, then we understand it as the task of transition between the corresponding objects, that is, the task of constructing corresponding software module.

A more complex feature of the categories under construction is objects with parameters. As parameters, you can use morphisms corresponding to some exponential. The entry $B(x)$ denotes object *B* with parameter *x*. Practically the record $B(x)$ denotes the whole family of objects generated by morphisms *x*. As a rule, we will use not arbitrary morphisms *x*, but morphisms corresponding to some exponential *A* (morphisms of type *A*). This will be denoted as follows:

$$B:(A=>\text{Type}).$$

Here is a parameterized object. *B* is itself a meta-morphism connecting object *A* and a special metaobject Type. In set-theoretic interpretation, Type is the set of all objects.

Two less traditional operations can be applied to objects with parameters: parameterized summation and multiplication. Operation summation is a generalization of the direct product operation and is denoted in a similar way:

$$x:A \,\&\, B\,(x).$$

When considering such an object as an interface, this entry denotes composite interface, the second part of which depends on the first one. In the set-theoretic interpretation, these are pairs whose sets of valid values of the second components depend on the values of the first ones. From the point of view of interfaces, this means that the component $B(x)$ cannot be contacted, not having a connection with the component $x:A$.

The multiplication operation is a generalization of the exponentiation operation and is indicated by:

$$x:A => B\,(x).$$

When considering such an object as an interface, this entry denotes an interface with a program whose interface depends on what is connected to the input.

These five specified operations allow categorically interpret the reasonings of positive logic, constructive logic without negatives (see [9, 10]). Note that the negations in the reasoning, leading to building programs mean that some situation is impossible and, therefore, it may not handle. But in terms of the reliability of the resulting software constructs it would be more adequate to speak not about the impossibility of the situation, but about its fallacy. Then, instead of negating *A*, you need to write the formula:

$$A\Rightarrow\text{Error},$$

where Error is a formula that means that a specific error has been detected, which must be properly processed. The results of such processing should be presented in the statement of the problem along with the results of an error-free work. Unlike the usual constructive logic from Error does not follow anything. You cannot construct such a morphism *f*, that for any object *A*

$$f:(\text{Error} \Rightarrow A).$$

The object Error is a normal object to be further processed. In practice you can consider many kinds of errors (see an example below).

The morphisms of the category in our case correspond to the proved sequents. Sequents are sentences used in logical conclusions, convenient for automation of reasoning. The task of proving the sequent turns into the task of constructing corresponding morphism.

Positive logic in some aspects most adequately reflects programming tasks. It allows deriving logically the correct programs for an environment in which there may be errors. Absence of negations guarantees consideration of all cases of work, since positive logic excludes nothing. Everything happens, and therefore it is necessary to reason consider all cases, even erroneous. Otherwise the reasoning will be incomplete. If we use machine support for correct reasoning, then it will not let us miss anything.

Categorical interpretation enriches the possibilities of positive logic in output of programs compared to set-theoretic interpretation. In the latter case, we build purely functional programs, that is, we automate programming in functional style. Using the same categorical interpretation, we can synthesize morphisms, realizing interaction between their initial and final objects, more complex than just getting representative of the target object on the representative of the initial object. Here you can, for example, synthesize the process of two-way interaction between objects related by morphism.

III. A SIMPLE EXAMPLE OF APPLYING THE DESCRIBED APPROACH

As a first example, consider a hypothetical communication system. The category of the subject area has as its main object *N*. It reflects the concept of communication node identifier. Since we will need to parameterize other objects of this category by identifiers of communication nodes, the object *N* will be considered an exponential:

$$N = (I\Rightarrow V).$$

In with this interpretation can consider, for example, that the identifier has an array structure. It may be number, name, code, identifying chain of actions, etc. In more complex cases the identifier can be a chain (or system) of actions that uniquely defines specific site of communication. Such an approach can be very useful for edtimating the complexity of generated algorithmic constructions. Statement $n{:}N$ means that *n* is an identifier of a communication node, that is, in our case, a morphism from the object *I* to the object *V*.

In addition, we assume that each node n corresponds to an object $B(n)$. It is interpreted as a potential base station that can be placed in the node *n*. Each of two nodes *m* and *n* corresponds to an object with two parameters $C(m,n)$, possible signal transmission channel from node *m* to node *n*.

The category of the domain also has four initial morphisms, initial communication system control modules:

1) morphism “bs” of searching available on channel $C(x,y)$ node y with base station $B(y)$ for node *x*:

$$\text{bs}:(x{:}N\Rightarrow y{:}N \,\&\, B(y) \,\&\, C(x,y)),$$

hereinafter, omitted brackets are implied to be grouped to the right,

2) morphism “bc” of the connection $C(x,y)$ of base stations $B(x)$ and $B(y)$ at nodes *x* and *y*:

$$\text{bc}:(x{:}N\Rightarrow B(x)\Rightarrow y{:}N\Rightarrow B(y)\Rightarrow C(x,y)),$$

3) morphism “rc” of inversion $C(y,x)$ of the connection $C(x,y)$ between nodes *x* and *y*:

$$\text{rc}:(x{:}N\Rightarrow x{:}N\Rightarrow C(x,y)\Rightarrow C(y,x)),$$

in other words, here we are talking about the organization of a duplex connection from existing simplex connection,

4) the morphism tr of the union $C(x,y)$ of the channels $C(x,y)$ and $C(y,z)$, the connections from the node *x* to the node *z* through the node *y*:

$$\text{tr}:(x{:}N\Rightarrow y{:}N\Rightarrow z{:}N\Rightarrow C(x,y)\Rightarrow C(y,z)\Rightarrow C(x,z)).$$

The problem to be solved here may be the problem of constructing a morphism of connecting arbitrary nodes:

$$\text{goal}:(x{:}N\Rightarrow y{:}N\Rightarrow C(x,y)).$$

This morphism is not difficult to construct from the original morphisms. Can see that connection occurs according to the scheme:

$$C(x,z) - B(z) - C(z,t) - B(t) - C(t,y).$$

Moreover, the formulas the specifications of the original morphisms are such that the logical derivation of the problems specification formula from the formulas of the specifications of initial morphisms in positive logic [9, 10] allows automatically build the desired morphism.

IV. A DIRECT SUM EXAMPLE

To illustrate working with direct sums, let's add another family of objects: $E(x)$. This is the error object at node *x*. Achieving this object may mean that the node x cannot work correctly for the problem being solved.

Then we replace the base station search module specification:

$$\text{bs}:(x{:}N \Rightarrow E(x) + y{:}N \,\&\, B(y) \,\&\, C(x,y)).$$

This, for example, may mean that nodes are sometimes outside the access area. The specification of the morphism of the solution also changes:

$$\text{goal}:(x{:}N\Rightarrow y{:}N\Rightarrow C(x,y)+E(\text{x})+E(y)).$$

It means that two nodes either can be connected, or at least one node does not work correctly. This morphism can also be constructed from the logical derivation of this goal formula.

V. LOOP MORPHISM EXAMPLE

Let us give an example of the specification of a dynamic programming cycle. Suppose that in the category of the domain there is some object *N*, corresponding to the concept of "product". This is an exponential whose morphisms are identifiers of some manufactured products.

Let there also be a family $R(x,y)$ of objects, where x and y are products, the object $R(x,y)$ corresponds to the concept of a method obtain product *x* from product *y*. For some pairs (x,y) such methods are impossible. The corresponding objects $R(x,y)$ in this case will be considered erroneous. Let in our subject area impossible endless paths of error-free transform of products. Then the family *R* generates a partial order on products.

Suppose there is another family of objects: the object $S(x)$ corresponds to some problem *S* solved for product *x*.

Under these conditions, the morphism of dynamic programming loop on products in the direction R to solve the problem S is specified as follows:

$$\text{for}:(n{:}N{=>}(i{:}N{=>}(j{:}N{=>}R(i,j){=>}S(j)){=>}S(i)){=>}S(n)).$$

Here n corresponds to the upper boundary of the loop, *i* corresponds to the loop parameter, *j* corresponds to the index of the accumulated solutions array. The meaning of the defined object defined here is as follows:

"If for each current product *i* of the fact that the problem *S* can be solved for all products preceding it (with respect to *R*) implies that it can also be solved for the current product, then it can be solved for any product at all."

Meaningfully this may be due to the fact that the relation *R* does not form endless or looping chains.

To illustrate the application of the loop morphism, consider the problem of determining the cost of the product. Let the object $S(x)$ correspond to the cost of the product *x*. Let all products be divided into two classes. The first class of products sre the products purchased on the side. Their cost is known in advance. The second class of products are the products made from other products. This is expressed by the presence of the following morphism:

$$\text{db}:(i{:}N \Rightarrow S(i) + j{:}N \,\&\, R(i,j)).$$

In fact, the morphism "db" is a database in which for each product either its known cost is contained, or a reference is made to another product and a method of producing the first product from the second. Another morphism required to solve the problem is a morphism that determines the cost of producing a new product:

$$\text{cn}:(i{:}N{=>}j{:}N{=>}R(i,j){=>}S(j){=>}S(i)).$$

It is a calculator that allows you to determine the cost of a new product by the identifiers of the manufactured and initial products, by the production method and by the cost of the original product. It is required to build a morphism that determines the value of any product in these conditions:

$$\text{goal}:(n{:}N{=>}S(n)).$$

The construction of this morphism is logically carried out by the following argument: To build morphism "goal" from a for morphism, it suffices to construct the missing argument:

$$\text{do}:(i{:}N{=>}(j{:}N{=>}R(i,j){=>}S(j)){=>}S(i)).$$

In fact, this argument is the body of the loop. It must be built for the loop to work. Then the morphism "for" leads from the morphism *n* of type *N* to the desired object $S(n)$. The morphism "do" must result from the morphism *i* of type *N* and the morphism:

$$\text{memory}:(j{:}N{=>}R(i,j){=>}S(j))$$

to the object $S(i)$. For this, morphism "db" is first applied to morphism *i*. It turns out to be the direct sum:

$$S(i) + j{:}N \,\&\, R(i,j).$$

Its first component is already a desired object. The second component remains:

$$j{:}N \,\&\, R(i,j),$$

which consists of the morphism *j* and the object $R(i,j)$. The morphism "memory" is applied to them. It turns out the object $S(j)$. Then the morphism cn is applied to the norphisms *i* and *j*, as well as to the objects $R(i,j)$ and $S(j)$. It turns out the desired object $S(i)$. Thus, the required morphism do is constructed. The problem is solved.

Note that the reasoning carried out here can be easily formalized in the logical inference systems of positive logic. In essence, this reasoning is a computer program corresponding to the constructed morphism. These considerations are amenable to automation with the use of appropriate systems for automatic search for logical inference. Thus, it gives the possibility of automatic or automated problem solving within the framework of the theoretical-categorical programming paradigm.

Note that in the previous example, the morphism of the loop operator was highly specialized. It was designed to solve only one problem *S*. To use more universal loops, we will need a transition to second-order logic. This can be illustrated by the following specification of a loop morphism:

$$\text{for}:(B{:}(N{=>}\text{Type}){=>}$$

$$n{:}N{=>}(i{:}N{=>}(j{:}N{=>}R(i,j){=>}B(j)){=>}B(i)){=>}B(n)).$$

Here the loop operator is not applied to the solution of one task *S*. This task is given by the parameter *B*. This is a family of objects indexed by morphisms *x*:*N*. The difference from traditional programming is that the problem to be solved here must be explicitly indicated, giving it as an argument of morphism. This is an analogue of the loop invariant. It must be set to start the loop. As the family *B* you can use, for example, the above task *S*. You can use complex tasks, for example, as the object $B(i)$ you can set

$$d{:}D{=>}T(i,d),$$

where the object D is an exponential that defines morphisms, corresponding, for example, to the concept of "cost type" (monetary, labor, energy, etc.); the object $T(x,d)$ represents the possible amount of expenses of the form d for making product x. Then the above specifications of the morphisms "db" and "cn" should change accordingly. For example, it could be like this:

$$\text{db}:(i{:}N \Rightarrow (d{:}D \Rightarrow T(i,d)) + j{:}N \,\&\, R(i,j)),$$

$$\text{cn}:(i{:}N \Rightarrow j{:}N \Rightarrow R(i,j) \Rightarrow d{:}D \Rightarrow T(j,d) \Rightarrow T(i,d)).$$

Generally speaking, the database may contain errors. For example, it may be cycling with respect to R. In this case, the loop operator helps to detect errors:

$$\text{for}:(B{:}(N \Rightarrow \text{Type}) \Rightarrow$$

$$n{:}N \Rightarrow (i{:}N \Rightarrow (j{:}N \Rightarrow R(\text{i,j}) \Rightarrow B(j)) \Rightarrow B(i)) \Rightarrow B(n)+\text{Error}).$$

Here the object "Error" signals detected errors. Then the specification of the problem being solved takes the form:

$$\text{goal}:(n{:}N \Rightarrow S(n)+\text{Error}).$$

Loop morphism on a finite domain N can be synthesized from a morphism of transitivity of the relation R:

$$trans:(x{:}N \Rightarrow y{:}N \Rightarrow z{:}N \Rightarrow R(x,y) \Rightarrow R(y,z) \Rightarrow R(x,z)).$$

This is written as:

metafor:

$$((x{:}N \Rightarrow y{:}N \Rightarrow z{:}N \Rightarrow R(x,y) \Rightarrow R(y,z) \Rightarrow R(x,z)) \Rightarrow$$

$$B{:}(N \Rightarrow \text{Type}) \Rightarrow$$

$$n{:}N \Rightarrow (i{:}N \Rightarrow (j{:}N \Rightarrow R(i,j) \Rightarrow B(j)) \Rightarrow B(i)) \Rightarrow$$

$$B(n) + x{:}N \,\&\, R(x,x)).$$

Here the object "Error" is instantiated as

$$x{:}N \,\&\, R(x,x).$$

This means that by mistake it turned out that something is produced from itself. A more complete specification of this universal transitive loop is as follows:

metafor:

$$(N{:}\text{Type} \Rightarrow R{:}(N \Rightarrow N \Rightarrow \text{Type}) \Rightarrow$$

$$(x{:}N \Rightarrow y{:}N \Rightarrow z{:}N \Rightarrow R(x,y) \Rightarrow R(y,z) \Rightarrow R(x,z)) \Rightarrow$$

$$B{:}(N \Rightarrow \text{Type}) \Rightarrow$$

$$n{:}N \Rightarrow (i{:}N \Rightarrow (j{:}N \Rightarrow R(i,j) \Rightarrow B(j)) \Rightarrow B(i)) \Rightarrow$$

$$B(n) + x{:}N \,\&\, R(x,x)).$$

Thus, we get a universal loop operator, independent of the specific objects being processed. The only global name for this specification is "Type".

VI. Recent related works

Recent works on the application of the ideas presented here include, for example, works [11] and [12]. In 2013, a unified framework for program synthesis problems was proposed by researchers at UPenn, UC Berkeley, and MIT [13]. Questions related to different approaches to the synthesis of program objects are theoretically considered in [14]: deductive, schematic and inductive synthesis of pure solutions, performed, in particular, in interaction with a man. Recent investigations in the area of logical-based program synthesis and transformations concern partial evaluation of programs [15], translation of an actor-based languages to the functional languages [16], verifying properties of time-aware business processes [17], synthesis software contracts from programs [18], synthesis of efficient generators for lambda terms [19].

References

[1] Curry H. B., Feys R. “Combinatory Logic” Vol. I. North-Holland, Amsterdam, 1958.

[2] Howard W. A. “The formulae-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, Boston, 1980, pp 479–490.

[3] Curien P.L. “Categorical combinatory logic”. *LNCS*, 1985; 194: 139-151.

[4] Crole R.L. “Categories for Types”, Cambridge University Press, 1994.

[5] Friedman H. “Equality between functionals”. In: *Logic Coll. '73*, 1975, pp 22-37, (*LNM* 453).

[6] Barendregt H. “Lambda Calculi with Types”, Handbook of Logic in Computer Science, Volume II, Oxford University Press, 1993.

[7] Cousineau G., Curien P. L., Mauny M. “The categorical abstract machine”. *LNCS*, Functional programming languages computer architecture. 1985; 201: 50-64.

[8] Kleene S.C. “Introduction to Metamathematics”, North-Holland Pub. Co., 1950.

[9] Church A. “Introduction to mathematical logic”, Vol. 1, *Princeton mathematical series, Annals of Mathematics Studies*, (Vol. 17), Princeton University Press, 1956.

[10] Rasiova H., Sikorski R. “The Mathematics of Metamathematics”. Panstwowe Wydawnictwo Naukowe, Warszava, 1963.

[11] Milewski B. “Category Theory for Programmers”. Version v1.0.0-0-g41e0fc3, October 21, 2018. https://github.com/hmemcpy/milewski-ctfp-pdf.

[12] Kulikov G.G., Antonov V. V., Fakhrullina A.R., Rodionova L.E.. “Method of structuring the self-organized intellectual system on the basis of requirements of the ISO/IEC 15288 standard in the form of the Cartesian closed category. (On the example of design of information and analytical system)”. In: *Proc of THE 20th INTERNATIONAL WORKSHOP ON COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES* (*CSIT'2018*). September 24-27, 2018, Bulgaria, Varna, 2018, pp. 135-139.

[13] Alur, Rajeev; al., et. "Syntax-guided Synthesis". Proceedings of Formal Methods in
Computer-Aided Design. 2013, IEEE. p. 8.

[14] Basin D., Deville Y., Flener P. Hamfelt, A., Nilsson J. F.: Synthesis of Programs in Computational Logic. In: Bruynooghe M., Lau K.-K. (Eds.) Program Development in CL, 2014, LNCS 3049: pp. 30–65.

[15] Alpuente M., Cuenca-Ortega A., Escobar S., Meseguer J.: Partial Evaluation of Order-Sorted Equational Programs Modulo Axioms. In: Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based
Program Synthesis and Transformation. LOPSTR 2016. Lecture Notes in Computer Science, vol 10184
(2017). Springer, Cham, pp. 3-20.

[16] Albert E., Bezirgiannis N., de Boer F., Martin-Martin E.: A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. In: Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based Program Synthesis and Transformation. LOPSTR 2016. Lecture Notes in Computer Science, vol 10184 (2017). Springer, Cham, pp. 21-37.

[17] De Angelis E., Fioravanti F., Meo M.C., Pettorossi A., Proietti M.: Verification of Time-Aware Business Processes Using Constrained Horn Clauses. In: Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based Program Synthesis and Transformation. LOPSTR 2016. Lecture Notes in Computer Science, vol 10184 (2017). Springer, Cham, pp. 38-55.

[18] Alpuente M., Pardo D., Villanueva A.: Symbolic Abstract Contract Synthesis in a Rewriting Framework. In: Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based Program Synthesis and Transformation. LOPSTR 2016. Lecture Notes in Computer Science, vol 10184 (2017). Springer, Cham, pp. 187-202.

[19] Tarau P.: A Hiking Trip Through the Orders of Magnitude: Deriving Efficient Generators for Closed Simply-Typed Lambda Terms and Normal Forms. In: Hermenegildo M., Lopez-Garcia P. (eds) Logic-Based Program Synthesis and Transformation. LOPSTR 2016. Lecture Notes in Computer Science, vol 10184 (2017). Springer, Cham, pp. 240-255.