Applicative computations and applicative computational technologies

Larisa Yusifovna Ismailova

National Research Nuclear University MEPhI (Moscow Engineering Physics Institute),
Kashirskoye Shosse, 31, Moscow, 115409 Russian Federation
lyu.ismailova@gmail.com

Abstract: This paper is aimed as a comparative study the main styles of applicative computations. This is a kind of computations where the objects could symmetrically act each other using the only metaoperation of application. The application, when done, results in a value which can be used as the object which can act with other objects on equal rights. This evaluation is considered in three main directions: 1) in mathematics, 2) in programming, and 3) in computing. It is shown that the evolution of the methods of evaluation the symbolic objects is moved to the modeling the interaction of objects as the interaction of corresponding information processes of reductions and/or expansions.
[Ismailova LYu. **Applicative computations and applicative computational technologies**. *Life Sci J* 2014;11(11):177-181] (ISSN:1097-8135). http://www.lifesciencesite.com. 25

Keywords: Objects, combinatory logic, applicative computational system, computational model, programming, computing

1. Introduction

This paper is aimed as a comparative study the main styles of applicative computations [1-3]. This is a kind of computations where the objects could symmetrically act each other using the only metaoperation of application. In symbolic notations this means that the pairwise combinations of objects are used, each pair resulting in a new object, which in turn could act on other objects. This is one of the dominant ideas in modern theoretical computer science [4]. In such a pair, the leftmost object is assumed as a function while the right neighboring object is the argument for this function. A result of applying the right object-function to left object-argument is assumed as a value of function on this argument. The mathematical study of symmetrical interaction of objects has a durable history [5-6].

We illustrate by example, what is the style and way of computing -- depending on the area of knowledge in which they are used. Consider three directions of evaluation: 1) in mathematics, 2) in programming, and 3) in computing. The first direction is discussed in details in [6], the second -- in [7-8], and the third -- in [4], [9].

As can be seen, in each case will be applied its own explanatory system and method of the graphic representation. However, it will be shown for example how, using the source combinator constants *K* and *S* to analyze the usual mathematical operations -- operations of the *composition*. As it turns out in terms of applicative computation, this operation is not elementary, i.e. non-atomic, and its "device" is defined by combining of the primary constants. Such an occurrence of "deep" structure of the composition operation, which is considered as atomic and indivisible from the viewpoint of algebra and mathematical analysis, emphasizes the expressive power of applicative computational systems. Of course, such methods can analyze the structure of other operations, but it is beyond the scope of this article, but the interested reader can find it in [8, 10] and other listed references.

2. Applicative computations

In information technologies (IT), we can rely on the information processes, but their diversity almost defies attempts at classification. However, in the 20-ies of the last century there was one discovery or invention, and this much depends on the position. It was discovered a few mathematical constants, using which you can construct or reconstruct all our accumulated mathematical knowledge [1].

These initial constants were called "combinators" that could be used as "building blocks", a kind of "bricks", of which the building of mathematics is constructed. The process of combining these special constants called "application operation" and a result of applying led to the formation of larger "building blocks", and the process of this enlargement could be iterated. For the development of mathematics and logic, it was not all just fun and instructive, but also rewarding, promoting the formation of the "mathematical constructivism". Arising from this mathematical calculus became known as "applicative computational systems" (ACS) [7], [8], [9].

2.1. Computation in mathematics

In mathematics and logic we operate with abstract objects, which for neutrality and giving the

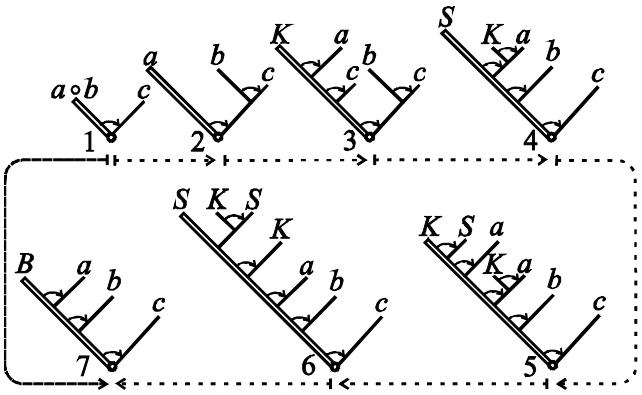

Figure 1. Characteristics of the composition combinator *B* and growing of tree computations for *B*, using trees for the combinators *S* and *K*. (*Explanation*. A transition from composition *a* ° *b* (unit 1) to representing combinator *B* (unit 7) is obtained by the counter-clockwise. This will require six steps, if there are only trees for computing *S* and *K*.

greatest generality called "obs" with corresponding ob-systems.

Computational environment of combinatorial logic (see [2]) is a highly symmetric in the sense that if you take two objects, *M* and *N*, then we can talk about the result of computation or evaluation, when *M* is applied to *N*, which is written by (*M N*). In this record, *M* plays the role of the left object, and *N* – of the right one. Such an object (*M N*) is binary in nature, by construction, and inductive classes of objects can be topologically represented by binary trees. In this computation, the first object *M* is observed as a function which is applied to the *N* as the argument and, in relation to the images of ordinary mathematics, it is about the formation of evaluation of *M* at the "point" *N*. But if in classical mathematics swap *M* and *N*, then construction "*N* from *M*" in understanding of the value of "*N* at *M*" is meaningless.

Within applicative system this computation is not only forbidden, but also has a definite meaning, since from the computational viewpoint the objects are absolutely symmetrical.

We are to introduce the computational characteristics of generic constant combinators *S* and *K*. When the characteristics are written equationally, using equality, we get

$$S\ a\ b\ c = a\ c\ (b\ c),$$
$$K\ a\ b = a.$$

We can show the process of "growing" tree computations for the composition operator in Figure 1. The composition of two functions *a* and *b* is defined by

$$a \circ b\ (c) = a\ (b\ (c)),$$

and in an applicative notation this is the conversion

$$a\ (b\ (c)) = a\ (b\ c) = B\ a\ b\ c.$$

In the original set of combinators there is no combinator *B* with these properties. But, as it turns out, such a combinator can be derived. The generation process is as follows in Figure 1.

Construct the tree computations associated with the composition (unit 1). The resultant tree is shown in the computing unit 7. If it has existed in the ob-system, the transition can be made to it counterclockwise. See if we can create a virtual track, giving it a tree computing which is constructed by combining the known tree computations. This tree can grow in stages, moving from tree to tree in a clockwise direction. Transition 1-2 is simply the application of the definition of the composition. Transition 2-3 corresponds to the conversion

$$a\ (b\ c) = K\ a\ c\ (b\ c),$$

which carried out in accordance with the computational, or combinatory characteristic of combinator *K*. Transition 3-4 corresponds to the conversion

$$K\ a\ c\ (b\ c) = S\ (K\ a)\ b\ c.$$

Similarly, the transitions 4-5 and 5-6 correspond to conversion

$$S\ (K\ a)\ b\ c = K\ S\ a\ (K\ a)\ b\ c,$$
$$K\ S\ a\ (K\ a)\ b\ c = S\ (K\ S)\ K\ a\ b\ c.$$

However, object *S (K S) K a b c*, arranged last in the chain of the virtual objects, corresponds to tree computations with a "canonical" serial arrangement of *a*, *b* and *c* along the branches (indicated by a single line in the figure). However, virtual object *S (K S) K* was in the "top" of the computation tree crown, forming it "trunk" (shown by double line). In other words, this virtual object has exactly the computational characteristics that required of object *B*, representing the composition (unit 7). This completes the round contour conversion clockwise:

the path from object *a* (*b c*) to object *S (K S) K* is found.

This method can be used in building the computation trees for many other mathematical objects, presenting them to the appropriate virtual objects and their corresponding properties of computational trees.

2.2. Computations in programming.

Let us now try to penetrate deep into the structure of the objects that at first glance appear substantially atomic and indivisible with respect to the transformations in which they participate. In particular, try to "split" -- of course, not materially but relative to the selected computational system -- the well-known operation of composition.

Suppose that there are objects *a*, *b* and *c*, in which, for example, the following meaning is embedded: *a*, *b* are operations *add1* of adding one, or "successors" *add1 z = z + 1*, and object *c* is equal to *3*. For composition of *a* and *b* using the argument *c,* we obtain

$$\begin{aligned} (a \circ b)\ c &\equiv (add1 \circ add1)\ 3 \\ &= add1\ (add1\ 3) \\ &= add1\ (4) \\ &= 5. \end{aligned}$$

This is a well-known meaning given to the operation of the composition and its internal structure is usually not concerned at all. The computational technology in use allows looking differently at normal operations, distinguishing their details of the internal structure. Try, for example, to answer the question whether the composition is the unit operation, or is generated as a derived object in the system of the generic objects-combinators. For this we take the object *S* (*KS*) *K*, which is composed of objects already known as combinators and computationally are understood. Use them to learn how this object can interact with other objects, i.e. determine its *combinatory characterization*. Initial configuration of the objects uses the first argument *a*, where the additional two arguments being analyzed are temporarily in parentheses: (*b*), (*c*). Reasons for this are as follows: *S* as in combinatorial computations showing arity equal to 3, the arguments *b*, *c*, in parentheses, do not act on S-computation directly.

Step 1-1: transition from S (*K S*) *K a to K S a* (*K a*). So combinator *S* is able to interact with three of his arguments – *K S*, *K* and *a*. Computation is distributed, and the first branch of the generated application is *K S a*, and the second branch -- *K a*. This is the *S*-reduction.

Step 1-2: transition from K S a (*K a*) *to S* (*K a*). Located in the first branch object *K S a* can be *K*-reduced, as the left-most object *K* has the object arguments *S* and *a,* which are necessary for transformation. Now a result of the interaction of objects is generated.

Step 2-1: transition from S (K a) b c to K a c (b c). Start with the generated object configuration. Interaction of object was performed as follows. As a result of *K*-reduction of the previous step, *S* is formed. It is known that distributor *S* forms its result by applying its left branch object to the right branch object. The result of this application *S* (*K a*) cannot be reduced further, as the left-most object *S* has the only first argument of *K a*.

Step 2-2: transition from K a c (b c) to a (b c). Object of the first branch of *K a c* as the left-most object contains combinator *K*, which has the necessary pair of arguments *a* and *c*, and performing *K*-reduction results in *a*.

The object of second branch is not reducible as there is no information about the structure of the corresponding leftmost object *b*, so now we have to perform applying the result of computation in the first branch to the result of computation in the second branch, i.e. applying *a* to *bc*. This is exactly *a* (*b c*), or commonly used infix notation (*a* ° *b*) *c*.

This shows that *S* (*K S*) *K* is a virtual representation of the composition operation.

Step 3: forming composition combinator B, which allows obtaining a (*b c*). Now we need to fix this semantic structure as a characterization, the elements of which have already been prepared. A representation of the composition is found and this is object *S* (*K S*) *K*. It remains to reduce references to this virtual object *S* (*K S*) *K* determining *B = S* (*K S*) *K*, which completes the process of reduction and leads to the needed characterization.

Briefly summarize the computational process in the direction of reduction for object *B*, which corresponds to perform the analysis of its potential internal structure. Computational analysis of the behavior of combinator *S* (*K S*) *K* gives the following:

$$\begin{aligned} S\ (K\ S)\ K\ \ a\ b\ c &= K\ S\ a\ (K\ a)\ b\ c \\ &= S\ (K\ a)\ b\ c \\ &= K\ a\ c\ (b\ c) \\ &= a\ (b\ c) \end{aligned}$$

The reduction above raises the question, but where was known in advance that object composition *B* has its virtual representation of *S* (*K S*) *K* in the world of interacting objects, generated by the objects-combinators *S* and *K*? In order to overcome this doubt, try starting from a result of composing objects *a* and *b*, denoted by (a ° b) c = a (b c), to generate the composer *B*. This means precisely that *B* interacts with a sequence of objects *a*, *b* and *c*, reducing in their composition *a* (*b c*).

Solving the problem of synthesis of an object with a predetermined characteristic of the

interaction requires *not* a reduction, but *expansion*. The symbolization process allows jotting down the entire chain of expansions, leading to a desired result.

The synthesis of combinator *B*, having a characteristic of composition is as follows:

$$\begin{aligned} a\ (b\ c) &= K\ a\ c\ (b\ c) \\ &= S\ (K\ a)\ b\ c \\ &= K\ S\ a\ (K\ a)\ b\ c \\ &= S\ (K\ S)\ K\ a\ b\ c \\ &\equiv B\ a\ b\ c. \end{aligned}$$

This computation is in the direction of *expansion*. Looking ahead, we note that it is enough to read the steps of analysis above in reverse order and each of the separate steps in reverse order as well, reading the equalities from right to left. In order to trace the expansion chain, the computations are dropped down by steps.

Step 1: the transition from a (*b c*) *to K a c* (*b c*). Use the result of the composition of objects *a* and *b* which is written as *a* (*b c*) for any argument *c*. This is the initial configuration of objects, and the target to be achieved as a result of expansion, is (*a* ° *b*) *c*, or, equally, *B a b c*. For purely formal reasons it is necessary simply to "disclose the parentheses", in which objects *b* and *c* are enclosed. The only way to do this is in using the *S*-expansion, but we need to pre-arrange a duplicate object *c*. And it is not difficult to ensure, using the *K*-expansion of the object *a*. This *K*-expansion process prepares the subsequent implementation of the *S*-expansion in accordance to its rule.

Step 2: transition from K ac (*b c*) *to S* (*K a*) *b c*. For disclosing parentheses we had to pay a duplicate object *c*. But this duplicate can be deleted, and for this we need to perform S-expansion.

Step 3: the transition from S (*K a*) *b c to K S a* (*K a*) *b c*. Again we need to open the parentheses, and they are significant in that enclosed application *K a*. This is done by *K*-expansion.

Step 4: transition from K S a (*K a*) *b c to S* (*K S*) *K a b c*. The result of the previous step was a duplicate of the object *a*, which need to be removed. This is done by S-expansion.

In the last step the target configuration objects is achieved. The object S (KS) K is synthesized and this object in its structure is a composer *B*.

2.3. Computation in computing

We show that the combinatorial *B* can be directly obtained by combining the *K* and *S*.

First of all, fix the object *a* (*b c*). The idea is that we need to release object *c* from the direct influence of the object *b*. Mathematically, this means that the need to open the parentheses. To do this we synthesize this distributed computation, creating another instance of *c*, which is achieved by the advent instance of combinator *K*. In symbols, this is as follows:

$$a\ (b\ c) = K\ a\ c\ (b\ c).$$

Further, one of the instances of *c* is to be eliminated, which requires the instance combinator *S*, but in the process the remaining instance of *c* is derived out of dependence on *b*:

$$K\ a\ c\ (b\ c) = S\ (K\ a)\ b\ c.$$

The same method is used to derive object *a* out of dependence on the object *K*. It is done in two steps. First, a second instance of object *a* is generated, distributing computation and generating the instance combinator *K*:

$$S\ (K\ a)\ b\ c = K\ S\ a\ (K\ a)\ b\ c.$$

The entire output chain is as follows:

$$a\ (b\ c) = K\ a\ c\ (b\ c) = S\ (K\ a)\ b\ c = K\ S\ a\ (K\ a)\ b\ c.$$

Now one of the two instances of object *a* is to be eliminated using the generated object *S*:

$$K\ S\ a\ (K\ a)\ b\ c = S\ (K\ S)\ K\ a\ b\ c.$$

Thus, the target object is synthesized, it remains only to put

$$S\ (K\ S)\ K\ a\ b\ c \equiv B\ a\ b\ c.$$

Thus, in a total, output chain looks as follows:

$$a\ (b\ c) = K\ a\ c\ (b\ c) = S\ (K\ a)\ b\ c = K\ S\ a\ (K\ a)\ b\ c = = S\ (K\ S)\ K\ a\ b\ c \equiv B\ a\ b\ c.$$

3. Conclusions and Future Work

1. The applicative interaction of objects in mathematics usually is not symmetric leading to "ordinary" functions with known number of their arguments before the evaluation. This restricts the real usage of arbitrary given functions.

2. The demands on arity of functions in applicative interaction of objects in programming are less restrictive. The functions can be used with partially known arguments which become actual ones on later stages of computation. This enables the bidirectional "information process" of analysis/synthesis of objects in computations "on fly".

3. Most of interacting symmetry is in general computation model, used in pure computing, where objects-actors can capture other objects along their applicative ability. This means that only the partially evaluated objects-processes can exist and interact with other partially evaluated ones.

This preliminary study gives rise to further study of the kinds of interaction environments. This could be done by restricting the applicative pre-structure of some, but not all the objects.

Acknowledgements:

This work is a generalization of the results, which are associated with the construction of conceptual and computational model obtained at

different times during the projects, partially supported by Russian Basic Research Foundation grants 14-07-00119-a, 12-07-00661-a, 14-07-00072-a, 12 -07-00646-a, 13-07-00716-a, 12-07-00554-a, 14-07-00054-a.

Corresponding Author:
Dr. Larisa Ismailova
Department of Cybernetics
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoye shosse, 31, Moscow, 115409, Russian Federation
E-mail: lyu.ismailova@gmail.com

References

1. Cardone F., Hindley JR. History of lambda-calculus and combinatory logic. In Logic from Russell to Church, volume 5 of Handbook of the History of Logic, Eds., Dov M. Gabbay and John Woods. Elsevier, Amsterdam, 2009;732–617.
2. Hindley JR, Seldin JP. Lambda-Calculus and Combinators, an Introduction. Cambridge University Press, 2008; 345.
3. Bimbó K. Combinatory Logic: Pure, Applied, Typed. CRC Press, 2012; 345.
4. Wolfengagen VE. Applicative computing. Its quarks, atoms and molecules, Ed. Dr. L.Yu. Ismailova. Moscow: Center JurInfoR, , 2010; 62
5. Rosser, JB. A mathematical logic without variables. Part 1. Annals of Mathematics, 1935; 36: 127-150.
6. Scott, DS. Outline of a mathematical theory of computation. Technical report, Oxford University Computing Laboratory Programming Research Group, 1970.
7. Wolfengagen VE. Combinatory logic in programming, Ed. Dr. Ismailova, L.Yu. Moscow: Center JurInfoR, 2003; 336.
8. Ismailova, LYu, Kosikov SV, Zinchenko KE, Mikhailov AI, Bourmistrova LV, Berezovskaya AV. Equationally Expressed Evaluation. In the Proceedings of the 9th International Workshop on Functional and Logic Programming, WFLP 2000, Ed. Maria Alpuente, Benicassim, Spain. September 28–30, 2000; 135-143.
9. Wolfengagen VE. Semantic modeling: computational models of the concepts. In the Proceedings of the 2010 International Conference on Computational Intelligence and Security, CIS 2010. Sponsors: Xidian University, Beijing Normal University, CPS of IEEE. Nanning, 2010; 42-46. DOI= http://dx.doi.org/10.1109/CIS.2010.16.
10. Homotopy Type Theory: Univalent Foundations of Mathematics. The Univalent Foundations Program, 2013. Institute for Advanced Study, 2013; 480. Date Views 05.02.2014 homotopytype theory.org/book/

6/24/2014