Applicative Approach to Information Processes Modeling
Towards a Constructive Information Theory

Viacheslav Wolfengagen, Vladimir Roslovtsev, Leonid Shumsky,
Artyom Bohulenkov and Artyom Sakhatskiy
Department of cybernetics, Moscow Engineering Physics Institute "MEPhI", Kashirskoe sh., Moscow, Russia

Keywords: Applicative Computational Environment, Information Process, Information Object.

Abstract: The goal of this work is to present an approach to constructive definition of information processes and objects. Our approach is based on applicative computational systems (ACS), so that both (information) processes and objects are formal entities in an ACS. We also outline the usage of π-calculus as an operational semantics for our constructive processes execution. The usage of the presented ideas in a distributed application development framework is outlined.

1 INTRODUCTION

Of the different definitions of the notion of a (computer) program one may be formulated in the way that a (computer) program is a system containing at least one of the following components:

- a *model* of information processes running in a domain and information objects participating in those processes;
- a system of information processes and information objects that *themselves* exists in that domain;
- an environment providing those objects' and existence and processes' execution.

In a computer program we cannot deal directly with 'physical' objects and processes such as (machine) parts and their assembling, so we have to create a model for those objects and processes to represent them in our program. On the other hand, some objects and processes (for example, software, digital images and movies sold and distributed through internet) are digital in their nature and thus may be themselves made part of our programs. And generally those objects and processes in a program cannot exists all alone, they require some support, an environment to allow their interaction, as well as lifecycle management, and the like.

Therefore, the understanding of what information objects and processes are, what is their structure and laws of existence – is essenssial.

It is worth noting that many of the problems occuring in software development are due to the huge gap that commonly exists between the problem damain's (conceptual) models and the conceptual system (semantics) of the programming language in use, if any such system is indeed clearly specified, which is rarely the case. It is beneficial, of course, to use the same conceptual system, or at least the same basic approach in constructing conceptual systems – the one for domain modelling and the one for the programming language.

The main idea of the present paper is that such a common basis may be found in applicative computational systems (ACS). ACS provide a constructive, compositional style of building objects; when augmented with an appropriate type system, abilities of (automatic) extraction of semantic information are drastically increased; interoperability problems are solved relatively easily; effective valuation techniques may be devised, including those with optimization features.

The rest of the paper is organized in the following way. Section 2 presents an introduction to ACS's basics – in a way suitable for the purposes of the paper. The sections 3 and 4 provide an outline for modelling information processes and information objects in a constructive way. Section 5 discusses connection with the π-calculus, section 6 discusses an implementation of stated ideas. Section 7 sums up the paper's main results.

Wolfengagen V., Roslovtsev V., Shumsky L., Bohulenkov A. and Sakhatskiy A..
Applicative Approach to Information Processes Modeling - Towards a Constructive Information Theory.
DOI: 10.5220/0004563303230328
In *Proceedings of the 15th International Conference on Enterprise Information Systems* (ICEIS-2013), pages 323-328
ISBN: 978-989-8565-60-0

2 A GENERALIZED MODEL FOR OBJECTS IN APPLICATIVE COMPUTATIONAL SYSTEMS

In applicative computational systems (ACS) objects are functional entities such that:

- the main way to construct composite objects is using the application operation (an object-function being applied to an object-argument);
- object's role (to be either a function or an argument) depends on the context in which the object appears, and an object, in an untyped system, may be applied to itself;
- objects-functions' arity is not, potentially, predefined and shows itself in the course of computations;

Following the idea from (Roslovtsev and Luchin, 2009), the class of objects in an ACS is constructed by induction on objects structure:

1. (induction basis)
   a) there is an infinite set of variables and a set (possibly, empty) of atomic objects;
   b) variables and atomic objects are atoms, and all atoms are objects;
2. (induction step) if $A_1, \dots, A_n$ are objects and σ is an n-ary term-forming function (TFF) then $\sigma A_1 \dots A_n$ is an object.

A TFF is a meta-operation used to construct complex objects from simpler ones. For example, application operation '$\cdot\cdot$' is a binary TFF (if A and B are objects then so is 'AB'). In λ-calculus there is another (binary) TFF – functional abstraction, which binds one variable in one object: if 'x' is a variable and 'A' is an object then '$\lambda x.A$' is also an object. In applied systems other TFFs may be defined, e.g. tuple constructors. The notation '$\sigma A_1 \dots A_n$' implies that the TFF σ is higher-order 'function' of a type of the form '$T_1 \to \cdots \to T_n \to T_{n+1}$'. The alternative is to write $\sigma(A_1 \dots A_n)$, with σ having a type of the form $T_1 \times \dots \times T_n \times T_{n+1}$. This second option, for example, illustrates that we do not usually deal with 'partially constructed' objects. The first approach, on the other hand, is more natural to ACSs and less demanding to the meta-theory's expressive power since there is no predefined notion of Cartesian product in ACS. For the purposes of the present paper it is unimportant which option to choose.

The question of choosing a set of atomic objects is a good one – even too good to be even tried to discuss it in any depth, much less to be answered, in the present paper. Two points, however, are worth noting. The first is that, actually, objects' atomicity is not part of their nature, it is not absolute, but, rather, it is a motivated assumption: further decomposition of such objects is either redundant or leads to inconsistencies (Wolfengagen, 2004). Secondly, though one may chose different suitable sets of atomic objects for different purposes, those sets would share a common computing foundation – for which combinators K and S are good candidates, see (Wolfengagen, 2010).

Since all objects of an ACS are built from a fixed set of atomic objects D using a fixed set of TFFs F, the generalized applicative prestructure may be written in the form of the pair:

$$(D, F)$$

In ACS, application operation exists by default and may be omitted in F. In case of combinatory logic with IKS basis that prestructure would look like $(\{\mathrm{I}, \mathrm{K}, \mathrm{S}\}, \{\})$, and for the pure (without any atomic objects included) λ-calculus: $(\{\}, \{\lambda \cdot\cdot\})$.

There is a very important kind of objects, called combinators. Combinator is an object that contains no free variables. In combinatory logic it means that a combinator is built out of atomic objects only, and in λ-calculus every variable occurring in that object is bound by a functional abstraction operation.

3 INFORMATION PROCESSES CONSTRUCTION

The basic idea underlying our process decomposition approach is that every process must be a combinator of some kind. That yields several advantages:

- every process is closed, does not reference any global variables and its output depends only on the data supplied explicitly to the process's input;
- because of that, too, and because a process is, more or less, an ordinary object, an existing process, be it simple or complex, may be re-used to define other processes;
- a process may be passed to another process as an argument, thus allowing computation adjustment at run-time;
- very powerful, yet relatively simple, process algebras may be formulated.

Generally, processing an input object comprises three stages. First, the input object is decomposed onto several components and each of them, secondly, is sent to a corresponding specialized (sub)process (which may be either a structural part of the given process, or an independent entity). Finally, the partial results yielded by the

'subprocesses' are aggregated in some way, e.g. 'wrapped' with a TTF into a resulting object.

One powerful approach in process construction is defining a set of 'standard' process 'schemes' to combine the already constructed processes into a more complex one.

Let $F: T_1 \to \cdots \to T_m \to T_{m+1}$ be a TFF. For every such TFF let us define a set of 'projecting' objects $\pi_F^i: T_{m+1} \to T_i$ (of course, with $i \in [1, m]$), such that:

$$\pi_F^i(Fa_1 \dots a_m) = a_i,$$

– for every suitable set of objects $a_1: T_1, \dots, a_m: T_m$. Finally, consider the currying 'operator' $\mathbb{H}^n$ with the combinatory characteristics:

$$\mathbb{H}^n f[x_1, \dots x_n] = f x_1 \dots x_n$$

Assume that $\langle f_1, \dots, f_n \rangle x = [f_1 x, \dots, f_n x]$ holds for every suitable (e.g. type-compatible, according to a certain type system of choice) set of objects $f_1, \dots, f_n$ and x. Then a process that takes an object $Fa_1 \dots a_m$ and yields another object $Gb_1 \dots b_l$ would in general look like this:

$$\mathcal{F} = \mathbb{H}^l \mathcal{G} \circ \langle g_1, \dots, g_l \rangle \circ \langle f_1 \circ \pi_{\mathcal{F}}^1, \dots, f_m \circ \pi_{\mathcal{F}}^m \rangle, \quad (1)$$

or

$$\mathcal{F} = \mathbb{H}^l \mathcal{G} \circ \langle g_1 \circ f_1, \dots, g_l \circ f_l \rangle, \quad (2)$$

where all f_i are subprocesses that perform a certain pre-processing of the input (on a by-component basis in case of (1), and in case of (2) – on a more general one). The responsibility of subprocesses $g_1, \dots, g_l$ is to yield individual components $b_1, \dots, b_l$ of the resulting object. Finally, $\mathbb{H}^l \mathcal{G}$ unpacks the tuple $[b_1, \dots, b_l]$ and produces the final result. Of course, both in (1) and (2) one may add explicit post-processing handlers. An essential point is that every component $b_1, \dots, b_l$ is evaluated independently of all others, which is rather handy in parallelized environments. If, on the other hand, some of these components depend on the others, an appropriate scheme may also be devised, but it is likely to be more complex and, obviously, yield a lower parallelization rate.

In (1) and (2) every $f_1, \dots, f_m, g_1, \dots g_l$ may be either an internal part of $\mathcal{F}$ or an external process (and may be replaced independently), or it may stand for a 'process variable' p_i ($i = [1, k]$), or contain its (free) occurrence(s). Functional abstraction of $\mathcal{F}$ by such variables will produce a schema:

$$\hat{\mathcal{F}} = \lambda p_1 \dots p_k . \mathcal{F}$$

Given already constructed processes $\mathbf{p}_1, \dots, \mathbf{p}_k$, one may construct a new complex process $\hat{\mathcal{F}} \mathbf{p}_1 \dots \mathbf{p}_k$. Note that, actually, the scheme $\hat{\mathcal{F}}$ may be replaced with another one, provided that all typing constraints are satisfied. Next, very naturally (most type systems would allow that) one can introduce variables ranging over schemes, thus leading to 'scheme constructors'.

Most of those schemes and scheme constructors are likely to be domain-dependent and the purpose of their introduction is to facilitate (automate, if possible) the construction of processes in such environments where:

- the number of processes is exceedingly high and/or processes structure is very complex and requires extensive decomposition;
- processes makeup and/or structure is unfixed by the domain nature and adjustments are to be made in a timely fashion.

The approach described in this section is partly inspired by D. Scott's flow diagrams, see (Scott, 1971).

4 BUILDING INFORMATION OBJECTS

Information objects are objects of a formal system representing entities from the 'real world' in computations and reflecting their attributes: characteristics and relations, the difference between these concepts being, strictly speaking, informal since the same attribute of the same entity might be called a characteristic or a relationship depending on the assumed point of view. The difference between ground types and concepts in description logics does not help in the matter. E.g. person's full name might be seen, and with benefit, as a characteristic, and yet its range would be a concept.

The relational approach is perfectly suitable for attribute representation but ultimately fails in reflecting entities' structure, being rather unnatural at this point. In pure relational systems the concept of computed attributes is represented via 'views' (i.e. relations that are named but not stored – actually, relation variables rather than relations, see (Date and Darwen, 2000) for detailed explanation). A more advanced and sound method is provided by some frame algebras (Roussopulos, 1977) and (Wolfengagen, 1984). But the structural aspect is still only mimicked via relationships.

In ACS the structural aspect is reflected very easily and naturally using TFFs in generally the same way as was shown with process schemes. The difficult part is to find suitable tools to represent attributes.

For computed attributes the obvious way is using a function – likewise to how the property construction in object-oriented languages like C# is implemented through methods. For non-computed attributes one has to yield a stored value, and there are two basic options: to use one single tuple to store all values related to the given object, or to use a distinct pair for every attribute. The pros and contras in each case are about the same as using highly normalized and denormalized relations in a relational environment, but we, at least presently, prefer the normalized version since it leads to conceptually more clear solutions.

Let $a{:}\,T$ be an object of type T in a ACS. This object is either atomic, or it is complex and build out of, in the end, atomic object using TFFs, as was explained earlier. A property (of objects of type T) is a function of type $T \to T'$, where T' is property's range. In case of a computed property it is an ordinary λ-expression, very likely, but not necessarily, referencing other object properties (the exact syntax is of no importance to us at this time). A non-computed property would reference a binary relation variable associated with the attribute to retrieve the relevant values.

Thus, we end up with a relational environment, not unlike the one described by E.F. Codd in (Codd, 1979). The most straightforward way of embedding relations in an ACS is conceptually simple. A relation may be thought of as a pair, with relation's header and body for the pair's elements. The header is a list of attributes, where every attribute is a domain- name pair; the relation's body is a list of tuples.

There are several ways to represent tuples in ACS, one of them being:

$$(a_1, \dots, a_n) = \lambda r.\, r a_1 \dots a_n$$

The usual way to represent a list is via nested pairs.

In conclusion to this section we note that both information processes and information objects are built likewise and follow the same rules. That means, in particular, that a process may have attributes, and, in a sense, an information objects may contain information processes as part of their definition – in the form of (computed) properties.

5 RELATED WORK: PI-CALCULUS

π-calculus is an extension of the standard calculus of communicating systems (CCS) allowing advanced process description: sending (and retrieving) named data blocks over named channels, the channels themselves being legal 'data blocks'. Every process in this theory is a sequence of subprocesses that either send or receive data blocks (purely computational processes are not considered). The main connectives to build complex processes out of simpler ones are: sequential execution, parallel execution and conditional branching.

In our approach processes are closed applicative objects (combinators), therefore we suggest using an extension of asynchronous polymorphic π-calculus as process execution semantics. That requires a certain extension of the standard 'chemical' model to formalize not only data exchange but processes' components execution as well. Also, such an extended model allows defining processes that involve data exchange between several systems.

Assuming (1) as process' scheme and $\mathcal{X}$ standing for a certain input object of $\mathcal{F}$, here below are stated the main process execution rules:

1. Every 'projection' mapping $\pi^i_{\mathcal{F}}$ is translated to $a_i(\mathcal{X}) \mid \left(\overline{1}\pi^1_{\mathcal{F}}\mathcal{X} \mid \dots \mid \overline{m}\pi^m_{\mathcal{F}}\mathcal{X}\right)$, i.e. execution of the process $\mathcal{F}$ begins with distribution, over a number of channels a_i, data required to get input object's components.

2. Every subprocess $f_1, \dots, f_n$ must not contain any data exchange actions, i.e. it must be a purely computational, closed (at least, within $\mathcal{F}$) process, in which case the following holds: $f_i \to i(d).\overline{i}f_i d$.

3. Every subprocess $g_1, \dots, g_l$ receives, as its input, a data block containing all the processed components of $\mathcal{X}$ and yields a certain component of the resulting output object. Accordingly,

$$g_i \to vt(1(y_1).\overline{t}y_1 \mid \dots \mid m(y_m).\overline{t}y_m \mid \\ \mid t(z_1 \dots z_m).\overline{o}g_i z_1 \dots z_m)$$

Note that such an execution scheme is possible in an asynchronous environment only.

4. Every process g_i returns it's result over the internal channel o; the 'last' stage of the process $\mathcal{F}$ awaits for all such partial results and then constructs the final result, e.g. using a certain TFF.

These rules outline the use of π-calculus as a mechanism underlying execution of our constructive processes.

6 IMPLEMENTATION CONSIDERATIONS

Based on the ideas presented in this paper, the authors are working on a framework (for Microsoft .NET Framework platform). This framework is

intended in providing flexible, extendable and scalable environment for process (and, of course, object) definition and execution. It is being implemented in the form of a distributed heterogeneous computational network (DHCN) consisting of nodes of several types:

- computational nodes that host processes;
- resource nodes, being (persistent) data storages;
- service nodes of various kinds;
- transport nodes that form a transport network to link together all other nodes and provide isolation and communication between different parts of the overall system.

Due to space limitations, we cannot present here a complete and detailed description of the framework's design and functioning. Briefly, DHCN's basic terminology is that of solving tasks: for every task defined in the system there are solvers somewhere in the network, hosted on computational nodes, capable to solve tasks of that particular type, actual input data being located on one or more resource nodes and the whole process being initiated from a client node (a kind of service node). Solvers playing one the central roles in the whole system, their development is facilitated by both requiring only to override a couple of abstract methods (in the decompose-compose style) and providing a set of solver composition schemas, each of which may be created in almost the same way as an ordinary solver.

Nodes' spanning over the physical network may be nearly arbitrary, of course, without solvers or clients being aware of the fact, thus making the framework suitable for (scalable) distributed systems development. Developing such systems initially was the main framework's application in mind, but presently higher-level interfaces, offering a conceptual basis of processes and components, are under design.

We conclude this section by noting that the main theoretical and engineering solutions regarding the framework under discussion are published in (Roslovtsev and Shumsky, 2012a) and in (Roslovtsev and Shumsky, 2012b).

7 CONCLUSIONS

In this paper we present a constructive approach to information processes and objects definition, which way being, in our opinion, beneficial in several ways. Our approach is based on applicative computational systems (ACS), so that both (information) processes and objects are formal entities in an ACS. We provide an extended model for objects in an applicative environment to facilitate processes and objects construction. We present a way of constructing information objects so that both their structure and properties being presented explicitly and soundly, and how that leads to integration of the applicative and relational paradigms. We also outline the usage of π-calculus as an operational semantics for our constructive processes execution. The usage of some the presented ideas in a distributed application development framework is outlined.

REFERENCES

Codd, E. F., 1979. Extending the Database Relational Model to Capture More Meaning. In *ACM Transactions on Database Systems*, Vol. 4, No. 4, 1979, pp. 397-434.

Date, C. J., Darwen, H., 2000. *Foundation for Future Database Systems. The Third Manifesto*. 2nd edition. Addison Wesley Longman, Inc.

Roslovtsev, V. V., Luchin, A. E., 2009. Concept of Higher-Order Applicative Computational Environment. In *Proceedings of the 11th international workshop on computer science and information technologies CSIT'2009*, pp. 48-53.

Roslovtsev, V. V., Shumsky, L. D., 2012a. Applicative Methods of Computational Process Decomposition. [In Russian.] In *Proceedings of the 3rd International Conference on Applicative computation Systems (ACS'2012)*. NEI Institute of Contemporary Education "JurInfoR-MGU", Moscow, Russia. pp. 224-233.

Roslovtsev, V. V., Shumsky, L. D., 2012b. Developing a Service Bus for Computational Process Distributing Environment. [In Russian.] In *Proceedings of the 3rd International Conference on Applicative computation Systems (ACS'2012)*. NEI Institute of Contemporary Education "JurInfoR-MGU", Moscow, Russia. pp. 258-265.

Scott, D., 1971. *The Lattice of Flow Diagrams*. Lecture Notes in Mathematics, Vol. 188. – Springer. – pp. 311-366.

Roussopulos, N. D., 1977. *A semantic network model of data bases*. Ph.D. Thesis. Dep. of Computer Science, University of Toronto.

Wolfengagen, W. E., 1984. Frame Theory and Computations. In *Computers and Artificial Intelligence*, Vol. 3, No. 1, 1984. pp. 3-32.

Wolfengagen, W. E., 2004. *Methods and Means for Computations with Objects. Applicative Computational Systems*. "Center JurInfoR", Moscow, Russia. [In Russian].

Wolfengagen, W. E., 2010a. *Applicative computing. Its quarks, atoms and molecules*. Edited by Dr. L.Yu. Ismailova. "Center JurInfoR", Moscow, Russia.

Wolfengagen, W. E., 2010b. The Parameterized Relational Model. Towards 40th Anniversary of Relational Data Model. In *Proceedings of the Second International Conference on Applicative Computational Systems*. NEI Institute of Contemporary Education "JurInfoR-MGU", Moscow, Russia.