On Theory of Names to be Used in Semantics of References in Functional and Object-Oriented Languages

Alexei I. Adamovich [1] and Andrei V. Klimov [2]

[1] *Ailamazyan Program Systems Institute of RAS, 4a Peter I str., Pereslavl-Zalessky, 152021, Russia*
[2] *Keldysh Institute of Applied Mathematics of RAS, 4 Miusskaya sq., Moscow, 125047, Russia*

Abstract

The long-standing problem of adequate formalization of local names in mathematical formulas and semantics of references in object-oriented languages taken on their own without objects, is discussed. Reasons why the existing approaches cannot be considered suitable solutions, are explained. An introduction is given to the relatively recent works on the theories of names and references of the group lead by Andrew Pitts. The concept of referential transparency, in which contextual equivalence is used instead of the usual equality of values, is analyzed. It is the main property, which these theories are based upon: such modified referential transparency is preserved when a purely functional language is extended with names and references as data. An outline of a constructive denotational semantics of the extended functional language is given. It is argued that the modified referential transparency, along with many other valuable properties, can be also preserved for mutable objects that change to a limited extent. This leads to a model of computation between functional and object-oriented ones, allowing for a deterministic parallel implementation.

Keywords

Bound variables, local names, references to objects, nominal set theory, referential transparency, contextual equivalence, monotonic objects, internally deterministic programs

1. Introduction

Consider the long-standing problem of adequate formalization of local names, bound variables in formal languages and references in object-oriented languages. In mathematics and programming, over the years, a certain style of working with names and references has been established, and it may seem that there is no problem here. However, in Section 2 we show that this is not the case: the methods of working with formulas with quantifiers presented in textbooks on mathematical logics (for example, [13]) contain unpleasant subtleties, artifacts of the popular formalization style. Another solution presented by N. Bourbaki in their Treatise [5] – a graph representation of the basic mathematical language – is by no means better, although instructive. It is discussed in Section 3.

A group of mathematicians led by Andrew Pitts has been developing the missing theories of names for almost thirty years [6, 11, 14–18, 20, 21]. We discuss their works in Section 4 and the idea of a purely functional language based on these theories, with the data domain extended with names, in Section 5. In the field of object-oriented programming, names turn into references to objects. But in this paper names and references are treated on their own without what they name or reference to. From the point of view of mathematical logics, such an extended functional language is nontrivial since it violates *referential transparency*.

In Section 6, we rectify this and show that if in the definition of referential transparency, the usual equality of values is replaced by *contextual equivalence* (a.k.a. *observational equivalence*, *Leibniz equivalence*), then such *modified referential transparency* holds for a functional language with names

SSI-2021: Scientific Services & Internet, September 20–23, 2021, Moscow (online)
EMAIL: lexa@adam.botik.ru (A.I. Adamovich); klimov@keldysh.ru (A.V. Klimov)
ORCID: 0000-0003-1392-8871 (A.I. Adamovich); 0000-0003-0418-7311 (A.V. Klimov)


or references. This observation is the cornerstone, which the theories developed by A. Pitts' group are based upon. The properties that are preserved and that are violated in the extended functional language are analyzed in Section 7.

In Section 8, we give an idea and outline of a constructive denotational semantics of the extended functional language. We hope that after reading this section, the feeling of mystery will disappear.

In Section 9, we list the steps that can (and should) be performed further from a functional language with names and references to objects with states—first immutable and then with restricted mutation – so that referential transparency based on contextual equivalence is preserved.

We also point out the practical significance of these works in the field of parallel programming: maintaining the contextual equivalence of the results of computation in any order means determinacy. In cases where a parallel program has the determinacy of both final and intermediate results of computation, it is much easier to debug, it is predictable and reliable. (Such strong form of determinacy is referred to as *internal*.)

Our contributions are as follows.

- We point out the problem of adequate formalization of names and references as data and argue that the works by Andrew Pitts' group have almost closed the problem for functional languages and opened the way to complete semantics of object-oriented languages, which has yet to be developed.
- We demonstrate that contextual equivalence, together with the modified referential transparency, is the cornerstone of the theories of languages with names or references as data.
- We present an outline of a constructive denotational semantics of the extended functional language.
- We discuss steps from A. Pitts' theories to a restricted object-oriented language that retain some of the nice properties of a functional one.

2. The problem of names and references as data in metamathematics and programming

In programming languages, the concept of a *name* appears in two seemingly different guises: *identifiers of variables* in program code and *references* to objects (Note that references and pointers arose in programming and made sense before the concept of an object was fully established in object-oriented languages).

Identifiers and references "live" in different times: variable identifiers – in *static time* from the moment when a program is written until it is converted by a compiler into executable code; references – in *run time* when a program is executed and does not change, while data are created and manipulated. These times overlap when it comes to *metaprograms*, programs that analyze and transform programs, where interpreters and compilers are the most known among such ones. In metaprograms, variables of ground programs are usually represented by references to some objects that store the necessary information. The question arises: Is this mapping of variables into references a purely programmatic trick or does it make some sense?

What about names in mathematics? Using the language of mathematics in everyday scientific activity, we differently perceive *variables*, *letters* as elements of a text on paper with which we work, on the one hand ("*static time*"), and *data domains*, from which variable takes values in some ideal mathematical "*run time*", on the other hand. They overlap in *metamathematics*, in logics, when the language of mathematics becomes an object of study, description, analysis and transformation. And in programming—in computer algebra systems, where a computer rather than a person manipulates formulas, "code" in mathematical languages.

Let us look at the part of the mathematical world where variables in formulas become an *object* of definition and manipulation.

Mathematical language is based on the fundamental concept of free and bound variables. For example, in the formula $\forall x\, \exists y\, P(x, y, z)$ the variables x and y are bound by the quantifiers $\forall$ and $\exists$, respectively, and z is a free variable, a parameter of the entire formula. It seems simple, but let us

open the classic textbook [13, p. 48] and see, for example, a definition of such an obscure (auxiliary!) notion:

> If A is a well-formed formula and t is a term, then t us said to be *free for* x_i *in* A if and only if no free occurrences of x_i in A lie within the scope of any quantifier $\forall x_j$, where x_j is a variable in t.

This phrase just means that the term t can be directly substituted for x in the formula A. Another common way to overcome the same problem is to provide a renaming requirement in the definition of substitution:

> When substituting a term t into a formula A, variables under quantifiers in formula A are renamed, if necessary, so that there is no conflict with the free variables in t.

For example, when substituting $f(x)$ for y in $\forall$x P(x, y) one needs to replace x with, say, z:

$$\forall z\, P\big(z, f(x)\big).$$

In addition, extra phrases such as “up to renaming variables”, “up to α-equivalence”, “fresh variable" are often found in reasoning, theorems and proofs about formal languages, which clutter up the code in proof assistants such as Coq and Agda.

Where do these problems come from? Our intuitive understanding of quantifiers doesn't seem to contain such “tricks”. Looking at a variable bound by a quantifier, we perceive it differently than a free one – not just as a letter. We imagine that the bound variable z in the above example is a completely different entity than the letter of the free variable x in it:

> The letter of a bound variable under the quantifier along with its occurrences in the scope of the quantifier, represents a certain entity that *differs* from the letters of all variables and all values in this formular and elsewhere in the mathematical world.

Intuitively thinking this way, we don't feel any need for renamings when substituting a term with such entities under quantifiers into anything.

Thus, the notion of a term free for a given variable, as well as the requirement for renaming when substituting, are just *artifacts* of a certain way of formalization of mathematics in common mathematical textbooks and have nothing to do with the essence of the idea of quantifiers and bound variables.

3. Solution by N. Bourbaki

Of course, this is not news. Let us recall the formalization of quantifiers given by N. Bourbaki [5]. It does not carry these artifacts and is closer to our intuitive understanding.

According to N. Bourbaki, quantifiers $\forall$ and $\exists$ with variables are not primary notions, but rather a readable representation of formulas in the basic language of mathematics introduced in Chapter 1 of Volume 1 of Treatise [5]. Interestingly, the basic language is not one-dimensional and is even more complex than a tree. Terms, formulas, program code, etc. in the traditional construction of formal languages are *trees*, and the transition from trees to graphs (for example, at the stage of name resolution in compilers) is a non-trivial step with a change in data representation.

In Treatise, the quantifiers $\forall$ and $\exists$ are defined through the terms of the form τ_xP(x) (with the informal meaning “such x that P(x) is true”), which in turn expand into a construct without the letter x. Let us show Bourbaki’s method of excluding variable letters, as if it were applied directly to the quantifiers, without going down to the level of τ-terms.

Encoding quantifiers in the style of N. Bourbaki, we replace the occurrences of bound variables with signs □ and draw lines to the corresponding quantifiers $\forall$ and $\exists$, depicting a link between them. Free variables remain letters. For example,

$\forall x\, \exists\, y\, P\big(x, y, f(x, z)\big)$ is encoded as $\forall\, \exists\, P\big(\square, \square, f(\square, z)\big)$.

What is instructive about this approach? Undoubtedly, N. Bourbaki sought to make the basic level of the language of mathematics as simple as possible. However, they made its representation not line-

ar, but graphical: the code in the base language is not a linear sequence of symbols but a graph, which in itself is far from a trivial concept. Such decisions are not accidental.

N. Bourbaki transferred the complexity here from another place. From where? From the substitution operation. Now the substitution of any term for any bound variable is defined without conditions like "a term is free for a variable" and other measures against the capture of free variables by external quantifiers.

Moreover, such representation reflects our intuition, mentioned above, that a variable bound by a quantifier is a completely different entity than a free variable: each bound variable is different from all other variables in the world because it is linked to the unique occurrence of a quantifier into a mathematical text. We can hypothesize that this was the decisive argument why N. Bourbaki transferred the complexity of the presentation of formulas to this level.

Back to programming. How do we encode Bourbaki's graph representation in an object-oriented language? The simplest solution is to keep the reference to the object representing the quantifier in the representation of each variable occurrence. This is how it is usually done in compilers after the name resolution stage: each occurrence of an identifier contains the reference to its declaration.

Such a solution based on a graph representation cannot be programmed in a purely functional language, bearing in mind that an adequate implementation should have the same computational complexity of transitions from a vertex to a vertex along an edge as in object-oriented languages, $O(1)$, as intuitively expected. The fundamental flaw of functional languages is that they do not provide means to efficiently represent general graphs, but only trees. The natural representation of a graph in functional languages is the list of vertices and arcs. Accessing such lists is more complex than $O(1)$. Moreover, this solution is complicated by the fact that it is necessary to assign unique identifiers to vertices and, perhaps, edges. There are no simpler solutions (except for the implementations of specialized libraries by lower level means).

In mathematics, the notion of a graph is defined in this way as well. Apologists of purely functional languages love them just for this—for their close correspondence to classical mathematics with set theory as a basis. On the other hand, object-oriented languages fall out of this mathematical picture. They provide effective means for adequately representing a variety of data structures, including graphs. This indicates unresolved problems in the foundations of mathematical semantics of programming languages.

Returning to the graph representation of N. Bourbaki's basic language, we see an interesting picture. It is based on the non-trivial notion of a graph, which is considered intuitive. Then, in set theory, it turns out that its concepts are not enough to adequately describe its own language (Note that the completeness/incompleteness of theories has nothing to do with this issue. In reasoning about the incompleteness, it is allowed to use arbitrary complex encoding of one theory in another; only the possibility of mapping is important. On the other hand, by the "adequacy" we mean the preservation of computational complexity).

These questions come up in programming when we perform analysis, program verification, and equivalent transformations preserving semantics. To do this, the semantics must be well defined, so that it is convenient and efficient to implement in proof assistants. Simply contemplating Bourbaki's representation with the words "it's intuitively clear that it works" is not enough.

4. Works on nominal methods

30 years ago, mathematician Andrew Pitts not only saw these problems of names and references (he was not the only one; see, for example, [7, 12] and [19, footnote 2[2] and Section 6.2]), but also began to solve them based on his mathematical experience in category theory and foundations of mathematics. Since then, together with his followers, he has built a rich theory of names – *nominal set theory* [6, 11, 14–18, 21], in a broader context also called *nominal methods*, *nominal techniques* [16], and

[2] Footnote 2 from [19]: "It is possible to postulate a data type of references (or pointers) **ref** θ, for every phrase type θ, whose values are storable in variables. This obtains the essential expressiveness that the object-oriented programmer desires. Unfortunately, our theoretical understanding of references is not well developed. So we omit them from the main presentation and mention issues relating to them in Section 6.2".

in several variants—in the form of set-theoretic models, in category theory, and within the operational approach.

In 2019, Andrew Pitts and Murdoch Gabbay received the ACM Alonzo Church Award for *their ground-breaking work introducing the theory of nominal representations* [1]. Papers [6, 14] are specifically mentioned in the award. Their achievements were highly appreciated by experts in logics and theory of computation, and now it is up to programmers to introduce these theories into the wide practice of program verification and semantics of programming languages. These results should form the basis of the theory of references and objects in object-oriented languages, which, as far as we know, has not yet been done.

5. Functional language with reference generator and its properties

To see the sources of problems and appreciate achievements, consider a purely functional language extended by a *generator of new names or references*. We use the later terms as synonyms, since references are names of objects, the authors of nominal methods have applied them to formulas and program code with local *names* and use this word, while the theory is well applicable to *references* to objects that interest us more. In language FreshML [15, 17, 20] developed by A. Pitts' group this operator is denoted by $\boldsymbol{new}$, as the instance creation expression in common object-oriented languages.

The notion of an object consists of 2 parts: the notion of a *reference* and the notion of the *state of an object*, where references can be talked about separately from states, but not vice versa. In addition, to deal with changing states, the concept of *computational time* is necessary. But while we are considering references alone, the concept of time is not required. In the language under consideration, where there are no objects with changing states, all the results of function calls do not depend on the order of computation, and time plays no role. Now we will deal only with the semantics of names and references as such.

Every evaluation of the $\boldsymbol{new}$ operator generates an atomic datum, the only property of which is to be *equal to itself* (*reflexivity* of equality) and to be *unequal to everything else*, that is, to all values in the data domain and all data generated by other calls to the $\boldsymbol{new}$ operators. Thus, although time does not matter to us, it is necessary to distinguish between individual evaluation actions.

Let us look at examples. Below we use the $\coloneqq$ sign in the following two roles:

- in our functional language, in the $\boldsymbol{let}$ statement to introduce a local name with a single evaluation of the right-hand side and to define functions;
- in the metalanguage of our reasoning as “equal by definition”.

For the rest, we use generally accepted mathematical notation, hoping that they will be understood without lengthy explanation.

Example. The following expressions in a functional language with the $\boldsymbol{new}$ operator evaluate to True:

$$\boldsymbol{new} \neq \boldsymbol{new},$$

$$\boldsymbol{let}\ x \coloneqq \boldsymbol{new},\ y \coloneqq \boldsymbol{new}\ \ \boldsymbol{in}\ x \neq y,$$

$$\boldsymbol{let}\ x \coloneqq \boldsymbol{new},\ y \coloneqq x \qquad \boldsymbol{in}\ x = y.$$

We see that re-evaluating the $\boldsymbol{new}$ operator produces unequal references and copying preserves equality.

Another example. The following function f produces a pair of unequal elements, because every time $f(x)$ is evaluated, the first element of the pair will be a new reference unequal to any other value in the world:

$$f(x) \coloneqq \langle \boldsymbol{new}, x \rangle.$$

Re-evaluation of f(x) always gives unequal results:

$$\forall x\, f(x) \neq f(x).$$

6. Referential transparency and contextual equivalence

The above examples point to the first (and main) problem of a functional language with the ***new*** operator: *referential transparency is broken*.

Referential transparency is a property of a term (expression, subexpression) in a given language that replacing any occurrence of a term with its value, evaluated only once, is an equivalent transformation. For a *referentially transparent language*, the following properties hold for all terms t and functions f that can be defined in the language:

$$t = t,$$

$$x = y \implies f(x) = f(y).$$

Referential transparency is so deeply rooted in the language of mathematics, mathematical logics that it seems difficult to approach the construction of a theory of a language, in which it is absent, without falling into the operational semantics with the notion of computational time. In particular, the denotation semantics of purely functional languages, which maps code of function definitions into functions of set theory, relies heavily on referential transparency.

Nevertheless, as the work of A. Pitts' group has shown, such a theory is possible, interesting, and productive. Its success is based on the following observation:

> The values obtained by multiple evaluation of a term in a functional language *L* with the ***new*** operator possess *contextual equivalence* (*observational equivalence*, *Leibniz equivalence*).

Two values x and y of some type D are *contextually equivalent* (denoting $x \approx y$) if they are indistinguishable by any predicate c, that is, a function of type $D \to Bool$, defined in language L:

$$x \approx y \coloneqq (\forall c \in L)\ c(x) = c(y).$$

It turns out that a functional language with the ***new*** operator has the following property.

For all terms t, functions f and values x and y we have

$$t \approx t,$$

$$x = y \implies f(x) \approx f(y)$$

and more than that

$$x \approx y \implies f(x) \approx f(y).$$

What we have obtained is a *modified referential transparency*, where equality of values is replaced with contextual equivalence.

Note that contextual equivalence cannot be checked within a language, there is no such computable relation as ≈, unlike the usual equality of values =. Contextual equivalence is a notion of the theory, the mathematical metalanguage.

7. Preservation and violation of functional language properties

A functional language with a reference generator ***new*** retains many of the good properties of purely functional languages, when equality = is replaced by equivalence ≈. Of course, when formulating equivalent transformations, it is necessary to keep track of which of these relations plays a role.

For example, the common subexpression (subterm) elimination is no longer an equivalent transformation in the general case (here the ⟹ sign means "equivalently converts to", rather than "implies" as above):

$$\ldots t \ldots t \ldots \implies \textbf{\textit{let}}\ x \coloneqq t\ \textbf{\textit{in}}\ \ldots x \ldots x \ldots$$

Nevertheless, such transformations can be used by imposing restrictions on the corresponding piece of code so that it does not go beyond the purely functional subset or go "not very much". Two examples of such restriction:

1. In case the language is statically typed: The type of term t does not use the reference type.

2. During evaluation of t, no new reference is generated (e.g., if there are no ***new*** operators in the code of function used in t) or new references do not fall into the value of t (which can sometimes be determined by static analysis of data dependencies).

Violation of the equivalence of the common subexpression elimination indicates that the classical construction of *denotation semantics* for functional languages, which assigns each language term its denotate, no longer works. According to the traditional denotational semantics, copies of the denotate of a term can be used as the result of evaluating all the occurrences of the term, which is not our case.

Nevertheless, the following properties are preserved in a functional language with references, albeit in a modified form:

- There exists the notion of a denotate of a term, which, although not a ready-made value, can be used instead of re-evaluating each copy of the term with a complexity not exceeding that of copying the denotate. This follows from the work of A. Pitts' group on set-theoretic models of nominal sets and the so-called *ν-calculus* [18]. The idea of such a denotate is presented in Section 8 below.
- Based on this, it is possible to give a denotative semantics of the extended functional language, which maps the code of each function definition to a set of pairs of a certain form, describing the relationship of subsets of arguments with denotations of the corresponding values. The idea of this construction is also shown in the next section.
- Due to the existence of such modified denotation semantics, the implementation of the extended functional language retains the possibility of *memoization* or *tabulation*: every time a given function is called, a pair $\langle argument, value \rangle$ is stored in a special table, a *memo*, and the $value$ is used in the subsequent calls when the argument is "congruent" to the stored one. Such a mechanism should not be used for functional languages in general, as it is time-consuming and memory-intensive, but it is justified for individual functions under the control of the programmer in cases where it can give a gain in efficiency.
- Last but not least, the *determinacy* of parallel computation inherent in purely functional languages is preserved, that is, the results obtained in any order of sequential or parallel computation of function calls are contextually equivalent. This mechanism is not always useful in general due to overly small granules of parallelism, but it can greatly increase efficiency under the programmer's control by specifying which function calls are beneficial to compute in parallel.

8. Constructive denotational semantics of a language with names

Let us show the essence of the idea by presenting an outline of a semantics of an extended functional language with references. The semantics is denotational in the sense that it assigns to each function definition and to each closed term a certain entity, which can be used instead of evaluating a function call. On the other hand, it is not a value, copies of which can be used instead of a term, as in the traditional denotational semantics. The construction is a little trickier. Below we follow our presentation at a conference [9].

Consider a pure functional language of your choice – be it statically typed (as Haskell) or untyped (as Lisp). A first-order language (without lambda expressions) is sufficient for our purposes, although there are no problems with a higher-order language either. Any of the usual operational semantics of the language is suitable.

Now extend the data domain with references that are atomic in the sense that each one has no other property than being equal to itself and unequal to other data, along with the ***new*** operator that generates a new reference each time it is called. The values passed as arguments and returned by functions are like the usual ones, but may contain references inside, e.g., a list of references is a first-class value.

The usual *set-theoretic denotational semantics*, where each term is assigned a single *denotation* that can be used in place of all occurrences of the term, does not work, because the result of a function call containing new references, e.g., that of term ***new*** in the simplest case, cannot serve as such a denotation. Moreover, the data domain with references is not a set in the sense of classical set theory. Of

course, non-standard set-theoretical models are possible, and they were indeed developed by A. Pitts' group [6, 11, 15, 16], but this greatly complicates the study.

Let the denotation of a closed term t not be the result of its evaluation y, as usual, but a term of the following form, obtained from y by considering as variables all new references $n_1, \dots, n_k$ produced during the evaluation of t and prepending the list of new references to y:

$$\nu\, n_1 \dots n_k . y.$$

Here ν is a special symbol (like λ in lambda expressions). Results of each evaluation action of the term t are obtained from this denotation by evaluating the following lambda expression:

$$(\lambda\, n_1 \dots n_k . y)\ \boldsymbol{new} \dots \boldsymbol{new}$$

which means this: substitute new references for all variables $n_1, \dots, n_k$ in y.

Let the denotation of a function f be a set of terms of the following form (rather than a set of argument-value pairs, as usual):

$$\lambda\, m_1 \dots m_l . x \mapsto \nu\, n_1 \dots n_k . y.$$

Such function denotation determines the result of evaluating a function call $f(a)$ as follows. Find the element of the denotation with such left-hand side x and such references $b_1, \dots, b_l$ occurring in a that $a = (\lambda\, m_1 \dots m_l . x) b_1 \dots b_l$. Then return the result by evaluating the following lambda expression

$$(\lambda\, m_1 \dots m_l . x)\big((\lambda\, n_1 \dots n_k . y)\ \boldsymbol{new} \dots \boldsymbol{new}\big)\ b_1 \dots b_l.$$

We see that this semantics satisfies the usual requirements for denotational ones: every term and function in a program has a denotation, although denotations are constructive terms rather than elements of the set-theoretic world. Therefore, it is referred to as *constructive denotational semantics*.

One can look at the function denotations from an operational point of view as they grow during computation. The denotations are gradually collected at each function call $f(x)$ by constructing an element of the denotation of f from the argument-value pair, where $m_1 \dots m_l$ are all references found in the argument x, $n_1 \dots n_k$ are the references generated during evaluation of the function call. It can be proved that the ordinary evaluation and the evaluation using these denotations give contextually equivalent results. This operational view shows the idea of tabulation or memoization that can be implemented for such languages.

9. What has been achieved and what's next?

Andrew Pitts, Ion Stark, Murdoch Gabbay and other authors [6, 11, 14–18, 21] developed several theories of functional languages with the name or reference generator based on set theory, category theory, as well as within an operational approach, and applied them to construct mathematical semantics of local names in formal languages and their equivalent transformations. These theories are richer and more convenient than the traditional reasoning about scopes, name shadowing, etc. In addition, they have implemented packages for the Coq and Agda proof assistants to automate constructing and checking proofs in terms of *nominal methods*.

The next step that has yet to be done is to build a complete theory of objects in object-oriented languages using a suitable theory of references. Existing theories are incomplete, as they leave questions of the exact semantics of references to naïve or semi-formal reasoning. As far as the authors of this article are aware, there has been no significant progress in the field of semantics object-oriented languages for decades, but this should happen after the ACM Award [1] has attracted attention of computer scientists to the achievements of A. Pitts' group.

Along this path, a theoretically interesting and practically important question is the construction of computational models intermediate between functional languages with a reference generator and general object-oriented languages. They can be created either by extending functional languages while retaining the selected properties, or by restricting object-oriented languages so that the necessary properties are fulfilled, or by a combining both.

Our main task here is to move along this scale as far as possible while maintaining the modified referential transparency based on contextual equivalence as a fundamental property, from which other

useful properties follow, albeit in a modified form. In particular, it is more reasonable to define the determinacy of parallel computation through the broader notion of contextual equivalence, rather than through the equality relation built into programming languages, since the indistinguishability of the results obtained in different orders of computation, rather than their equality, matters from a practical point of view.

Currently, the following stages of loosening restrictions and adding notions are observed:

- As a starting point, take a purely functional language with a reference generator, but without the concept of a stateful object. This level is discussed above.
- Add the notion of an *immutable object*. At first glance, there is little practical benefit from this, except that the reference equality is more efficient, it is checked much faster than comparing the content of objects (of course, provided that it corresponds to the meaning of the task). However, there is both a theoretical and a practical purpose of equality by reference: it can be defined for *all* data, including those that previously have no reasonable computable equality. For example, the functional values generated by lambda expressions, *closures*, are of this type. Some languages (e.g., Haskell) forbid comparing closures based on the idea that the equality of functions is algorithmically undecidable. However, closures are trivially equivalent if they are copies of the result of the same evaluation of a lambda expression – similar to how equal references go back to the same evaluation action of the operator ***new***. Such equality can always be defined for all data. Its only flaw is the lack of referential transparency. The special role of such equality is not accidental: it is *initial* in the sense of category theory, that is, the strongest relation entailing any other equivalence relation.
- Next, we allow *mutation of the states* of objects, but only those that are invisible from the point of view of the "observer" (from program code in this language), that is, the side effects are such that any order of computation produces *contextually equivalent results*. An open problem: Is it theoretically possible to characterize object classes defined in an object-oriented language that retain contextual equivalence for all (sub)expressions in the language? The general answers to these and similar questions seem to be negative, since here we are faced with algorithmic undecidability. But, as is usually the case both in mathematics and in programming practice, it is often possible (and necessary) to identify and study interesting special cases for which something meaningful can be said.
- Such a rich special case are objects that can be described as changing *monotonically* on some (*semi*)*lattice* [10]. It turns out that modified referential transparency based on contextual equivalence is maintained by a certain discipline of operations on objects, which we refer to as *monotonic objects*. This issue is investigated in [2–4, 8, 9]. An open problem: How to mathematically describe a lattice for a *monotonic class* declaration in an object-oriented language.
- In terms of *monotonic objects*, it is necessary to study how to *solve practical problems* that are not possible to implement in purely functional languages as efficiently as in object-oriented ones. First of all, the main flaw of purely functional languages is to overcome, which is that only trees can be manipulated in them, while graphs have to be encoded with less efficient data structures then in object-oriented languages, where vertices are naturally represented by objects, and the transition from a vertex to a vertex along an edge is a cheap atomic operation "dereferencing". It turns out that this problem is solvable [4].
- Next, we want to get the most out of computational models that are close to object-oriented but retain valuable properties of functional programs. *Analysis, transformations, verification, parallelization of programs* and *metacomputation* in general – all these kinds of program manipulation are more effective for languages with no or limited side effects. This is clearly seen in the greater or lesser successes of program specialization, such as *partial computation* and *supercompilation*, for different classes of languages. In particular, the *tabulation* or *memoization* mechanism, implemented based on the denotational semantics described in Section 8, can be used to implement *dynamic partial computations* [8].

The present authors are working on such model of computation, intermediate between functional and object-oriented ones, and a Java-like language that satisfies the above properties, with the practical goal of implementing a deterministic parallel programming system that will free the programmer

from the torment of debugging non-deterministic parallel programs on a fairly wide class of problems [2–4].

10. Acknowledgements

The authors are very grateful to the participants of the seminars at which this material was presented and thoroughly discussed: Igor Adamovich, Arkady Klimov, Yuri Klimov, Alexander Konovalov, Antonina Nepeivoda, Sergei Romanenko. Special thanks to Sergei Meshveliani who made valuable comments at the conference presentations.

11. Conclusion

We have considered the long-standing problem of adequate formalization of local names in formal languages in both mathematics and programming languages, as well as the problem of formalization of the semantics of references in object-oriented languages. Considering the history of the issue, motivations and solutions, we came to the conclusion that these are one and the same problem, and the seeming difference is that these concepts refer to different kinds, times, periods of activity, which have well-established names in programming: we deal with names in formal texts in *static time*, when a mathematical text or program is written and processed, e.g., by a compiler; and manipulation of data with references happens “in dynamics”, during *run time* of a program on a computer and when mathematical statements are evaluated in some ideal mathematical world. These periods intersect when a mathematical text becomes an object of manipulation on a computer, in computer algebra systems, in logics and metamathematics, and programs undergo deep analysis and transformation in *metacomputation*. An adequate theory of names and references is wanted.

Over the past decades, Andrew Pitts' group has made significant progress in the construction of such theories [6, 11, 14–18, 20, 21], for which they recently received the ACM Award [1]. They built models of *nominal sets*, that is, data domains with names or references, within the frameworks of set theory and category theory, and also studied them operationally. They proposed a purely functional language extended with names or references as data, called FreshML [15, 17, 20], and explored its semantics in their theories. We argue that these results will serve as a theoretical basis for constructing a complete formal semantics of object-oriented languages and developing methods for verifying programs with references and mutable objects.

It was demonstrated that the success of building these theories and the expectation of their fruitful development up to mutable objects are based on the *modified referential transparency*, in which the *contextual equivalence* is used instead of the usual equality of values. In particular, the notion of determinacy of parallel computation essentially requires contextual equivalence of results in different orders of computation, rather than equality of values. Based on such weakened referential transparency, we are able to extend purely functional languages with references, and then even with mutable objects subject to certain restrictions, while retaining many of the valuable properties of functional programs.

In the last section we presented our view of the possible stages of extending purely functional languages to object-oriented ones, first with references without objects, as in the works of A. Pitts’ group, and then with mutable objects under certain restrictions that ensure the preservation of the modified reference transparency and the properties resulting from it. Some results have been already obtained; others remain open problems for future work. What is most valuable in this development is the emergence of a model of computation and programming languages between functional and object-oriented ones (which the present authors work on [2, 3]). In these model and languages, the scope of practical problems efficiently and adequately solved is wider compared to purely functional languages, while many of their valuable properties, including deterministic parallel computation, are preserved. Among these tasks, graph manipulation [4] is the most important one, because in purely functional languages, efficient data are only trees.

12. References

[1] ACM Special Interest Group on Logic and Computation, Winners of the 2019 Alonzo Church Award, 2019. URL: https://siglog.org/winners-of-the-2019-alonzo-church-award.

[2] A. I. Adamovich, And. V. Klimov, How to Create Deterministic by Construction Parallel Programs? Problem Statement and Survey of Related Works, Program Systems: Theory and Applications 8 4(35) (2017) 221–244. https://doi.org/10.25209/2079-3316-2017-8-4-221-244.

[3] A. I. Adamovich, And. V. Klimov, An approach to deterministic parallel programming system construction based on monotonic objects, Vestkik SibGUTI (3) (2019) 14–26. URL: http://vestnik.sibsutis.ru/uploads/1570089084_1278.pdf.

[4] A. I. Adamovich, And. V. Klimov, Building Cyclic Data in a Functional-Like Language Extended with Monotonic Objects, in: V. Zakharov, N. Shilov, I. Anureev (Eds.), X Workshop PSSV: Program Semantics, Specification and Verification: Theory and Applications: Abstracts, A.P. Ershov Institute of Informatics Systems, Novosibirsk, Russia, 2019, pp. 11–19. URL: https://persons.iis.nsk.su/files/persons/pages/tezisy_seminara_pssv.pdf.

[5] N. Bourbaki, Theory of Sets. Springer, Berlin, Heidelberg, 2004. https://doi.org/10.1007/978-3-642-59309-3.

[6] M. J. Gabbay, A. M. Pitts, A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing 13 (2002) 341–363. https://doi.org/10.1007/s001650200016.

[7] C. A. R. Hoare, Record Handling, ALGOL Bulletin 21 (1965) 39–69. https://doi.org/10.5555/1061032.1061041.

[8] And.V. Klimov, Dynamic Specialization in Extended Functional Language with Monotone Objects, ACM SIGPLAN Notices 26:9 (1991) 199–210. doi:10.1145/115865.376287.

[9] And.V. Klimov, On Semantics of Names in Formulas and References in Object-Oriented Languages, in: S.A. Abramov, L.A. Sevastyanov (Eds.), Computer algebra: 4th International Conference Materials, CA'21, MAKS Press, Moscow, Russia, 2021, pp. 73–76. URL: http://www.ccas.ru/ca/_media/ca-2021.pdf. https://doi.org/10.29003/m2019.978-5-317-06623-9.

[10] L. Kuper, R. R. Newton, LVars: Lattice-Based Data Structures for Deterministic Parallelism, in: Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'13). Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326.

[11] S. Lösch, A. M. Pitts, Denotational Semantics with Nominal Scott Domains, Journal of the ACM 61:4 (2014) Article 27, 46 pages. https://doi.org/10.1145/2629529.

[12] B. J. MacLennan, Values and Objects in Programming Languages, SIGPLAN Not. 18:12 (1982) 70–79. https://doi.org/10.1145/988164.988172.

[13] E. Mendelson, Introduction to Mathematical Logic (2nd ed.), D. Van Nostrand Company, New York, NY, 1976.

[14] A. M. Pitts, Nominal Logic, a First Order Theory of Names and Binding, Information and Computation 186:2 (2003) 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X.

[15] A. M. Pitts, Nominal Sets: Names and Symmetry in Computer Science, Cambridge Tracts in Theoretical Computer Science 57 (2013), Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9781139084673.

[16] A. M. Pitts, Nominal techniques, ACM SIGLOG News 3:1 (2016) 57–72. https://doi.org/10.1145/2893582.2893594.

[17] A. M. Pitts, M. J. Gabbay, A Metalanguage for Programming with Bound Names Modulo Renaming, in: R. Backhouse, J.N. Oliveira (Eds.), Mathematics of Program Construction, MPC '00, volume 1837 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2000, pp. 230–255. https://doi.org/10.1007/10722010_15.

[18] A. M. Pitts, I. D. B. Stark, Observable Properties of Higher Order Functions that Dynamically Create Local Names, or: What's new?, in: A. M. Borzyszkowski, S. Sokołowski (Eds.), Mathematical Foundations of Computer Science 1993, MFCS '93, volume 711 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1993, pp. 122–141. https://doi.org/10.1007/3-540-57182-5_8.

[19] U. S. Reddy, Objects and Classes in Algol-Like Languages, Information and Computation 172:1 (2002) 63–97. https://doi.org/10.1006/inco.2001.2927.
[20] M. R. Shinwell, A. M. Pitts, M. J. Gabbay, FreshML: programming with binders made simple, SIGPLAN Notices 38:9 (2003) 263–274. https://doi.org/10.1145/944705.944729.
[21] I. D. B. Stark, Categorical Models for Local Names, Lisp and Symbolic Computation 9:1 (1996) 77–107. https://doi.org/10.1007/BF01806033.