CONTINUITY AS COMPUTABILITY

N. N. Nepeivoda

Ailamazyan Program Systems Institute, Russian Academy of Sciences, Pereslavl-Zalessky, Russia

✉ nnn@nnn.botik.ru

Abstract. The relationship between computability and continuity is studied. Computability over an arbitrary initial basis of data types and functions (a base) is considered using McCarthy recursive schemata and strongly typed operators of finite types. In this case, computable operators are proved to be strongly continuous in the Baire sense: for parameter functions with any value of the other arguments, it is possible to find a finite collection of their values that uniquely determines the result. Based on relative computability, an approach to constructive topology is developed within which the pointwise approach (an element is a fundamental sequence of neighborhoods) and the approximation approach of abstract topology (a function over topological spaces is a neighborhood relation) are equivalent. The concept of **B**-spaces is formulated, allowing one to constructivize separable spaces with a countable base of neighborhoods. The continuity of pointwise constructive functions of **B**-spaces and their transformability into neighborhood relations are proved. The equivalence of the concepts of computability relative to a certain base and continuity is established. The concept of a relatively constructive function is formulated: such a function transforms each element into an element constructible relative to its argument and a fixed base. Its equivalence to the concept of a countably continuous function formed by the union of a countable family of functions continuous on subspaces is established. Since any separable space with a countable base can be described as a **B**-space, this result contains no constructive restrictions. The connection between the proposed approach and other approaches to constructive topology is discussed.

Keywords: relative computability, separability, countable base, pointwise spaces, abstract topology, approximation approach, Baire continuity, continuity on subspaces.

INTRODUCTION

In research devoted to computability in topological spaces, its connection with continuity is necessarily present; see the discussion in Section 3. Computability is considered on the basis of a uniform standard model, i.e., Turing machines. The question arises about the relationship between the abstract general concept of computability and continuity. It will be addressed below.

1. ALGORITHMIC FORMALISM

1.1. Recursive Schemata

Programming practice has shown that functional programs for discrete and continuous tasks are written in different styles. The standard definition of computability in computer science using Turing machines or λ-calculus [1] is pragmatically correct primarily for discrete tasks. Although almost all variants of computability considered in theory are formally equivalent, their areas of practical applicability may differ. Since computability here will be analyzed for continuous tasks, we should be careful in choosing an appropriate definition of computability for the programming style used. In this paper, the main concept of computability is the relative computability of McCarthy recursive schemata [2] over a *base:* the basis of initial functions and predicates. It is generalized to the strongly typed calculus of operators of finite types. Formally, this is a restriction of the definition of λ-operators of finite types with a fixed-point combinator, direct products, and conditional operators, which was studied in detail in computer science [1, 3, 4].

Definition 1 (a tower of data types $\mathbb{T}$).

1. A finite collection of data types $\mathbb{T}_0$ that necessarily includes the **bool** type.

2. All types from $\mathbb{T}_0$ belong to $\mathbb{T}$.

3. A type $(\tau_1,\dots,\tau_n)$ is called a *record*. If $\tau_1,\dots,\tau_n \in \mathbb{T}$ and are not records, then $(\tau_1,\dots,\tau_n) \in \mathbb{T}$. If all members of a record are types from $\mathbb{T}_0$, then this record belongs to the subclass $\mathbb{T}_1$.

4. If $\tau_1,\dots,\tau_n \in \mathbb{T}$, then $(\tau_1,\dots,\tau_n \to \tau) \in \mathbb{T}$. Such types are called *functional*.

5. Each type is assigned an infinite collection of variables of that type.

Definition 2 (a base).

1. Each data type from $\mathbb{T}_0$ is assigned with a non-empty data set that necessarily contains the "soft error" element $\perp$. The **bool** type is assigned the set $\{T, F, \perp\}$.

2. For any types of the form $(\tau_1,\dots,\tau_n)$, there are functions

$$join:(\tau_1,\dots,\tau_n \to (\tau_1,\dots,\tau_n)),$$
$$pr_i:((\tau_1,\dots,\tau_n) \to \tau_i),$$

satisfying the condition

$$pr_i(join(a_1,\dots,a_n)) = a_i \text{ for } 1 \leqslant i \leqslant n. \qquad (1)$$

3. A finite collection of initial typed functions with arguments and values from $\mathbb{T}_1$ and constants with types from $\mathbb{T}_0$. (The logical constants **true** and **false** are present in any case.) All functions are stable relative to $\perp$:

If $f(x_1,\dots,x_n,\perp,y_1,\dots,y_k) = z$ and $z \neq \perp$,

then $f(x_1,\dots,x_n,x,y_1,\dots,y_k) = z$ for all x.

(Such functions are called *generic* in computer science.) Constants from the base are called *initial*. Tuples whose elements are all constants are also considered constants.

Functions with a logical result value are called *predicates*. ♦

Hereinafter, a sequence of expressions $a_1,\dots,a_n$ will often be denoted by $\vec{a}$. Accordingly, $\vec{\perp}$ is a sequence of $\perp$. Condition (1) means that the type of a record is isomorphic to the direct product of the value sets of the types that compose it. But the situation is somewhat more subtle: the value sets of functional types are not defined and, accordingly, there are no constants of these types until explicit definitions appear, with the exception of the initial functions. Therefore, condition (1) means that if some values of the corresponding types exist, then there is also a value corresponding to their *n*-tuple.

Furthermore, there are no restrictions on the appearance of $\perp$ as elements of a record. Obviously, the functions *join* and pr_i are stable relative to $\perp$. Hence, Clarke's warning can be avoided by considering only functions with one argument or with one subject argument and one functional argument.

Hereinafter, the *join* function will be omitted whenever no ambiguities occur.

Definition 3 (terms).

1. The constants and variables of type τ are terms of type τ.

2. If f is a term of type $(\tau_1,\dots,\tau_n \to \tau)$ and $t_1,\dots,t_n$ are terms of types $\tau_1,\dots,\tau_n$, respectively, then $f(t_1,\dots,t_n)$ is a term of type τ. (Such a term sequence $\vec{t}$ is called *suitable* for *f*.)

3. If b is a term of the **bool** type and r and u are terms of type τ, then

if b **then** r **else** u **fi**

is a term of type τ. Such terms are called *conditional*. ♦

The designation $t\{f_1,\dots,f_n,x_1,\dots,x_k\}$, abbreviated as $t\{\vec{f},\vec{x}\}$, means that only the listed constants and variables, except for the initial ones, are used in the term. Note that some of the variables may be of functional type. The *parameters* of a term $f(t_1,\dots,t_n)$ are all t_i that do not have the form $(r_1,\dots,r_k)$, and all r_i for such records. Due to type restrictions, this definition is not recursive. The meta-designation $f < t_1,\dots,t_n >$ means that $t_1,\dots,t_n$ is a list of all parameters of the term, rearranged in a convenient order. The order in the expressions $f < r_1,\dots,r_n >$ does not change in the same sentence.

Definition 4 (a recursive schema). Let $f_1,\dots,f_n$ be new constants of functional types called recursive functions. Then a collection of definitions of the form

$$f_i(x_1,\dots,x_k) \leftarrow t[f_1,\dots,f_n,x_1,\dots,x_k],$$

where $\vec{x}$ is a sequence of variables suitable for f_i, is called a *recursive schema*. ♦

The order of definitions within a recursive schema makes no sense.

Definition 5. A *computational environment* defined by a recursive schema consists of a base and functions defined by this schema (recursive functions).

1.2. Abstract Computability

Consider programs operating based on recursive schemata; let us rigorously define their execution and computational capabilities.

A term is called *closed* if it does not contain variables.

We will define a computational process in the spirit of term rewriting rules to make a computational system a subsystem of recursive operators of finite types from [1]; and the results established there will be utilized accordingly.

For fundamental reasons (operators must work not only with the arguments defined by them), computations may involve constants for all elements of types from $\mathbb{T}_0$.

Definition 6. A *normal form* is a term to which no transformation rule can be applied. The process of computing a closed term is defined as a sequence of subterm replacements until obtaining a normal form (if this ever happens). If a computation process does not terminate, then the value is $\perp$. A term is *computed* by applying one of the rewriting rules to it. A term *disappears* in the cases specified in the rewriting rules, or when it is a subterm of a disappearing term.

1. If a term has normal form, then the process is complete, and this normal form is its value.

2. If a term has the form $f(\vec{t})$, where f is a basic function and all parameters are constants, then it is replaced by the result of computing f.

3. If a term has the form $f<\vec{c},\vec{t}>$, where f is a basic function, $\vec{c}$ are constants, and $f<\vec{c},\vec{\perp}>\neq\perp$, then it is replaced by the result of computing $f<\vec{c},\vec{\perp}>$, and all elements of $\vec{t}$ disappear.

4. A term $pr_i(\mathit{join}(t_1,\ldots,t_n))$ is replaced by t_i, and all other t_j disappear.

5. **if true then** r **else** u **fi** is replaced by r, and u disappears.

6. **if false then** u **else** r **fi** is replaced by r, and u disappears.

7. **if** $\perp$ **then** r **else** u **fi** is replaced by $\perp$, and r and u disappear.

8. If a term has the form $f(\vec{t})$, where f is a recursive function, then it is replaced by the definition of f, with the variables therein replaced by the terms from $\vec{t}$.

The result of transformations is *intermediate* if only rule 8 can be applied to it.

Remark 1. The genericness condition essentially means that the given argument $\perp$ is not used in computations. Rule 3 expresses that this condition is valid for all initial functions.

Consider the structure of intermediate results.

Let us define a computation discipline that guarantees a result if it is possible.

Definition 7. A computation is *regular* if it satisfies the following constraints.

1. Rule 8 applies only in an intermediate term.

2. In the term **if** $f(t_1,\ldots,t_n)$ **then** r **else** u **fi**, the terms r and u and their subterms are not computed.

3. In an intermediate term in the subterm $f(t_1,\ldots,t_n)$, where f is not the initial function, t_i are not computed.

4. Any subterm $f(t_1,\ldots,t_n)$ appearing in an intermediate term either disappears in further computation or is computed. ♦

Definition 8. A recursive function *is total* if any of its regular computation terminates. A function *is general*[1] if it is total and any of its computation with arguments not equal to $\perp$ does not yield $\perp$. A recursive operator is *total* (*general*) if, under any substitution of total (general) functions into this operator instead of its functional parameters, the resulting function is total (general).

A function f of the type $\vec{\tau}\to\pi$, where all $\tau_i,\ \pi\in\mathbb{T}_1$, is *computable* if there exists a recursive function with $f(\vec{c})$ as its result computed for each value $\vec{c}$ of the arguments of f. An operator $\varphi<\vec{f},\vec{x}>$, where $\vec{x}$ and the result have types from $\mathbb{T}_1$ and $\vec{f}$ are of the types $\vec{\tau}\to\pi$, where all $\tau_i,\pi\in\mathbb{T}_1$, is *computable* if there exists a recursive operator Φ such that when extending the base by any $\vec{f}$ of suitable types and substituting $\vec{c}$ for $\vec{x}$, the result of computing $\Phi(\vec{f},\vec{c})$ is equal to $\varphi(\vec{f},\vec{c})$. ♦

Remark 2. In total functions, an error appears as a calculated value and can be processed. If a function loops, being calculated infinitely, then such a situation is generally impossible to process.

Definitions for operators are not limited to functions defined in the base. They must work on any functions. Also, note that any external function is total.

We do not attempt to define computability for operators of higher types, since the set of functions of such types remains uncertain.

1.3. Computability Properties

Theorem 1 (on correct computations).

1. *All terminating computations yield the same result.*

2. *If a computation terminates, then all regular computations terminate as well.*

P r o o f. Item 1 (the first statement of this theorem) is a consequence of the theorem on correct rewriting rules for typed λ-calculus with recursion; see [1, *Ch.* 5].

[1] This term is given by analogy with the general recursive function.

Item 2 is a consequence of the theorem on head recursion [1, *Sect. 3.7*].

Lemma 1. *No computable operator can produce a new function as a component of the result.*

P r o o f. According to syntactic restrictions, all normal forms are constants.

Since it is impossible to compute a new function, we define the concept of function transformation using a computable operator.

Definition 9. An operator $\Psi < f, x >$ transforms a function f into g if, after adding the function f to the base, $\Psi < f, c > = g(c)$ for all c . ♦

This definition is naturally generalized to the transformation of a tuple of functions into a tuple of functions.

Lemma 2. *Any computable operator* Ψ *transforms computable functions into computable ones.*

P r o o f. Let us add the definitions of the argument functions $\vec{f}$ to the definition of the operator Ψ. Without extending the base, we obtain the definition of the result function:

$$\psi(\vec{x}) \leftarrow \Psi(\vec{f},\ \vec{x}).$$

Definition 10. A finite collection X of values of the arguments $f_i(\vec{b}_i) = d_i$ of an operator $\Psi < f, x >$ *guarantees* a value y on c if, when substituting any functions satisfying $f(\vec{b}_i) = d_i$, the value is $\Psi < f, c > = y$. If any of its subsets does not guarantee the value, it *guarantees* it *exactly*. We will define the collection X as a set of pairs (c_i, d_i) (called a function *fragment*). A function has a fragment X if it takes the values d_i on all c_i. ♦

A practical criterion for checking that a fragment exactly guarantees a value is to test the operator with the following functions: F equal to $\bot$ everywhere except for a given set, that yields d_i on c_i, and its modifications that additionally yield an error on one of c_i. But this is not a universal solution: if a recursive schema loops, we will not receive a response. This can also happen during an "illegal" call to a function parameter, as a result of which the calculation proceeded a different path.

The restrictions imposed (the absence of initial operators) allow us to establish Baire's continuity of computable functions, which is violated in a more general case.

Theorem 2 (the first continuity theorem). *For any closed term* t *whose recursive schema contains only functional variables* $f_1, \ldots, f_n$ *and whose regular computation is finite, there exist fragments* X_i *such that when substituting other functions with the same fragments into it, the term's value does not change.*

P r o o f. The computation contains only a finite number of calls to parameter functions. If other functions have the same values on the given parameters, the computation will be repeated with the same result.

Example 1. This example illustrates the significance of type complexity restrictions for the base. The continuity theorem is violated if we add the original operator of the second type, $FZ: ((\mathbf{nat} \to \mathbf{nat}) \to \mathbf{bool})$, with the definition

$$FZ(f) \leftarrow \begin{cases} T \text{ if there exists no } i \text{ such that } f(i) = \bot \\ \bot \text{ otherwise.} \end{cases} \blacklozenge$$

Thus, the above concept of computability is adapted to relativization to many initial spaces and any initial functions, giving the strong continuity of computable operators of the second type.

Example 2. The minimal base generating standard computable functions of natural numbers is the type system $\{\mathbf{nat}, \mathbf{bool}\}$, where **nat** is interpreted as the type of natural numbers, constant $0 \in \mathbf{nat}$, a predicate $Z \in (\mathbf{nat} \to \mathbf{bool})$, and functions S, Pd of the type $(\mathbf{nat} \to \mathbf{nat})$, interpreted as

$$Z(x) \equiv (x = 0), \quad S(x) = x + 1,$$

$$Pd(x) = \begin{cases} \bot \text{ if } x = 0 \\ x - 1 \text{ if } x > 0. \end{cases}$$

Recall that truth and falsehood are always added.

Summation and multiplication are defined by the recursive schema

$$\begin{cases} A(x,\ y) \leftarrow \mathbf{if}\ Z(y)\ \mathbf{then}\ x\ \mathbf{else}\ A(S(x),\ Pd(y))\ \mathbf{fi} \\ M(x,\ y) \leftarrow \mathbf{if}\ Z(y)\ \mathbf{then}\ 0\ \mathbf{else}\ A(M(x,\ Pd(y)),\ x)\ \mathbf{fi}. \end{cases}$$

The second-order function I iterates the application of the argument f, x times, Err formally always gives an error; in fact, its execution never ends, and the yield is nothing:

$$\begin{cases} I(x,\ y,\ f) \leftarrow \mathbf{if}\ Z(x)\ \mathbf{then}\ y \\ \qquad\qquad\qquad \mathbf{else}\ I(Pd(x),\ f(y),\ f)\ \mathbf{fi} \\ Err(x) \leftarrow Err(x). \end{cases}$$

Lisp logic connectives [5] are defined by schemata, so we will use them freely below:

$$\begin{cases} A \wedge B \leftarrow \mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ F\ \mathbf{fi} \\ A \vee B \leftarrow \mathbf{if}\ A\ \mathbf{then}\ T\ \mathbf{else}\ B\ \mathbf{fi} \\ \neg A \leftarrow \mathbf{if}\ A\ \mathbf{then}\ F\ \mathbf{else}\, T\ \mathbf{fi}. \end{cases}$$

If natural numbers are present or simulated in the environment, we use standard designations for operations and identify natural numbers with their code.

1.4. Properties of a Generalized Graph

As noted above, we can consider operators with two arguments—the initial type and the functional type—without loss of generality. The following convention allows using the results of standard algorithm theory.

Convention on enumerability and lists. *For each initial type, an equality predicate and an enumeration function, denoted by en, are added so that the sequence $en^n(a_0)$ enumerates without repetition all elements of this type, where a_0 is a constant. At least one of the initial types is infinite.*

Then this infinite type can be used as an isomorphism of natural numbers. Let the *i*th element be simply denoted by i. Of course, if there are natural numbers among the initial types, they can be used directly. As in standard computability theory, we define primitively recursive encodings of *n*-tuples, lists, and finite sets by numbers, the union and intersection operations of encoded sets, and the predicate of belonging of a given number to a set. For convenience, we specify the encoding of a set as a list without duplicate elements.

Definition 11. *A generalized graph* is an enumerable set of triples (X, c, d), where the value c does not contain[2] the constant $\perp$, and for any triples (X_i, c, d_i) with the same c none of X_i is nested in another X_j. A generalized graph is a *graph of an operator* $F(f, x)$ if in all triples X is a fragment ensuring d on c. A graph is *complete* if for any φ, c for which the computation of $F(\varphi, c)$ terminates, there exists (X, c, d) in which X is a fragment of φ.♦

We define, in the form of a specification implemented by standard programming methods, an operator for computing an operator $F(f, x)$ on a complete graph G:

$$\begin{cases} \text{If the enumeration of } G \text{ has a triple } (X, x, d), \\ \qquad \text{where } X \text{ is a fragment of } f, \text{ then } d; \\ \text{otherwise, one operates infinitely.} \end{cases}$$

Thus, a complete generalized graph is a strong and adequate definition of an operator. It would be great if a recursive schema for it could be constructed using syntactic transformations of the operator schema. Consider the difficulties arising in this case.

Lemma 3 (call tracing). *Let a recursive schema define an operator Φ with functional parameters f_i and a subject parameter x (i.e., from $\mathbb{T}_1$). Then:*

1. When computing other functions of this schema, f_i can be applied only if the given function is also an operator and f_i is substituted as one of its functional parameters.

2. All places of potential application of f_i in a term t can be found statically by analyzing the recursive schema, and they have the form $pr_j(F)(t)$, where F is a functional parameter of some function.

3. The application of f_i will occur if and only if the place of potential application of f_i is found on the path of computing the parameter $pr_j(F)$ with the value of f_i.

Proof. Let us outline the proof. Since f_i is neither an initial function nor a function defined in the schema, it can only be the value of some parameter. This proves item 1 (the first statement) of Lemma 3.

Next, we construct a list of functional parameters of the schema for Φ whose components can be replaced by *f*. This list includes f_i itself. If, inside the definition of a certain function, a function from Φ is substituted for the parameter *g*, then *g* is added to Φ. Applications of parameters from Φ are potential applications of f_i. Item 2 of this lemma is established. Finally, item 3 follows from item 2. ♦

The traceability lemma also works in the case when a schema contains operators computing functions, since they cannot compute new functions. Subsequent constructs can, in principle, be carried out for the case when such operators are allowed. But this is not essential for the main objective of the paper (operators on functions of a topological space are not considered here). Therefore, the analysis will be restricted to second-order schemata that do not define functions yielding functions.

Task 1. Modify the definition of an operator so that when terminating the work on given parameters, it would yield a fragment of the parameter function that exactly guarantees the result.

Task 2. Modify the definition of a term with one functional variable so that it would take a fragment as a parameter instead of a function and, if possible, yield not only the result but also a diagnosis of whether the fragment is sufficient for correct computation.

Let *f* and $T[f]$ denote a function parameter and a term computed, respectively. Note that the schema transformations are performed for this particular term since preliminary tracing is necessary. Based on tracing, in Task 1, we can label the subterms in the original recursive schema; in Task 2, we can label the subterms of the transformed term and those of the definitions in the original recursive schema as well.

Definition 12. A subterm is *red* if it contains a call to f. A subterm is *yellow* if it does not contain such a call but is a direct component of a red term (an argument of a red function, a component of a red record, or an alternative of a conditional expression with a red condition). A subterm is *white* in all other cases. If $T[f]$ is white, we label it as yellow. ♦

[2] Recall that the element $\perp$ does not turn an entire record into an error.

Using the traceability lemma, one can color the subterms of any term.

It is more convenient to process the error indicator first, so it is always equipped with the **true** flag. Hereinafter, $\varnothing$ indicates an empty set; $\varnothing$ stands for an *n*-tuple of empty sets; finally, $\{x\}$ means a singleton. Without limiting generality, assume that all functions defined in a schema yield a record with n components. If S is a record, then (x, S) is the result of adding a component x to it. This does not lead to ambiguities: in a type tower, records cannot be components of records. Note that the standard folding of an *n*-tuple into a list code is not always correct due to Clarke's warning: a list code containing an error as one of its elements is always an error. To avoid unnecessary effort while dealing with particular numbers, we define the function $tail((x_1,\dots, x_n)) = (x_2,\dots, x_n)$ for each type of *n*-tuples encountered in a schema.

Let us construct the following auxiliary functions using standard methods:

1. the function $Ev(X, a)$, which yields $(\mathbf{false}, b)$ if $(a, b)\in X$ for some b, and $(\mathbf{true}, A)$ otherwise;

2. the function $\cup$, which constructs the union of two sets.

We denote by A a constant that will replace the value unnecessary below.

Modification 1. $t^p[X, t]$, computation with guaranteed success. It is obtained by adding a subject parameter X to all operators defined in a schema that contain potential applications of *f*, and by recursively replacing all calls $f(r)$ to the function parameter, traced using Lemma 5, with $tail(Ev(X, r^p))$. This modification provides the simplest partial solution to Task 2 without diagnosing errors associated with a parameter missing in a fragment. Note that even this partial solution is individual for $T[f]$ since the coloring depends on T.

Modification 2. Tracing and collecting the applications of a parameter functions: processing T into T^v.

For each function with a red entry, we define its red variant with an additional parameter X_1 of the fragment type.

1. The red variant of f:

$$f^v(x) \leftarrow \mathbf{if}\ pr_1(x)\ \mathbf{then}\ (\mathbf{true},\ pr_2(x),\ A)$$
$$\mathbf{else}\ (pr_1(Ev(X, tail(tail(x)))),\ X_1\cup\{pr_2(x)\}, \quad (2)$$
$$pr_2(Ev(X, tail(tail(x))))\ \mathbf{fi}.$$

2. The red variant of the remaining initial functions:

$$\varphi^v(X_1, x) \leftarrow \mathbf{if}\ pr_1(x)\ \mathbf{then}\ (\mathbf{true},\ pr_2(x),\ A)$$
$$\mathbf{else}\ (\mathbf{false},\ pr_2(x),\ \varphi(tail(tail(x))))\ \mathbf{fi}.$$

3. The red variant of the function, defined as $\varphi(\alpha, x)\leftarrow t,$ has the form

$$\varphi^v(X_1, \alpha, x)\leftarrow t^v,$$

where t^v is computed recursively according to the rules specified below.

We recursively make the following substitutions.

1. All white subterms remain unchanged. We replace all yellow t with

$$t^v = (\mathbf{true}, \varnothing, t).$$

2. The record $(t_1,\dots, t_n)^v(t_1,\dots, t_n)^v$:

$$(\mathbf{if}\ pr_1(t_1^v)\ \mathbf{then}\ (\mathbf{true},\ pr_2(t_1^v)\cup\dots\cup pr_2(t_n^v),\ A)\ \mathbf{elif}\dots$$
$$\mathbf{elif}\ pr_1(t_n^v)\ \mathbf{then}\ (\mathbf{true},\ pr_2(t_1^v)\cup\dots\cup pr_2(t_n^v),\ A)$$
$$\mathbf{else}\ (\mathbf{false},\ pr_2(t_1^v)\cup\dots\cup pr_2(t_n^v),$$
$$tail(tail(t_1^v)),\dots,\ tail(tail(t_n^v)))\ \mathbf{fi}.$$

3. The conditional term $\mathbf{if}\ b\ \mathbf{then}\ r\ \mathbf{else}\ u\ \mathbf{fi}^v$:

$$\mathbf{if}\ pr_1(b^v)\ \mathbf{then}\ (\mathbf{true},\ pr_2(b^v),\ A),$$
$$\mathbf{elif}\ pr_2(b^v)\ \mathbf{then}\ (pr_1(r^v),$$
$$pr_2(b^v)\cup pr_2(r^v),\ tail(tail(r^v))),$$
$$\mathbf{else}(pr_1(r^v),\ pr_2(b^v)\cup pr_2(u^v),\ tail(tail(u^v)))\ \mathbf{fi}.$$

4. The red function $\varphi(s)$:

$$\mathbf{if}\ pr_1(s^v)\ \mathbf{then}\ (\mathbf{true},\ pr_2(s^v),\ A)$$
$$\mathbf{else}\ (pr_1(r^v), pr_2(b^v)\cup pr_2(u^v),\ tail(tail(u^v)))\ \mathbf{fi}.$$

This complex program restructuring corresponds to the concept of a *continuation* in functional programs [1, 6]. Unlike the works cited, we do not add a third-level operator here, managing to restructure the function with level reduction.

Theorem 3 (generalized graph). *Based on the schema of a general operator* Φ, *one can construct a definition of a function that enumerates its generalized graph.*

P r o o f. Let us make an additional modification to the schema T^v for the term $T=\Phi(f_0, x_0)$, where $\Phi(f_0, x_0)$ are constants.[3] We introduce a new variable for a fragment X and add it as a parameter to all definitions and applications of functions , without using it anywhere inside the definitions (except for the mandatory parameter when calling any function). The only exception is (2), where it is

[3] Here, f_0 can be either a new initial function or one defined in the same schema; this affects the process of constructing T^v but not new modifications.

used in Ev. In this regard, we redefine the original operator Φ as

$$\Phi^g(X,\ x) \leftarrow \Phi^v[X,\ x].$$

With standard methods, we construct a certain enumeration $Enum(i)$ of pairs $(X,\ x)$ (a fragment and a subject parameter) starting from zero.

Let us define a function for constructing the initial segment of the generalized graph. For brevity and clarity, we write the frequently occurring expression $\Phi^g(pr_1(Enum(i)), pr_2(Enum(i)))$ simply as $[\Phi^g]$:

```
$Gr(i, G, n) \leftarrow$ if $i = n+1$ then $G$
    elif $pr_1([\Phi^g])$ then $Gr(i+1, G, n)$
    else $Gr(i+1, G \cup \{(pr_2([\Phi^g]),$
        $pr_3([\Phi^g]),\ pr_2(Enum(i)))\}, n)$ fi.
```

It adds, step by step, the found fragments that verifiably guarantee $pr_2(Enum(i))$ on $pr_1(Enum(i))$. In the limit, we obtain the complete graph of the operator.

Remark 3. This algorithm produces an enumeration with repetitions. This can be eliminated using standard methods.

A complete study of the relationships between operators and generalized graphs requires separate consideration. Here, we merely mention that the syntactic transformation of an arbitrary operator into a graph is generally impossible: its existence would mean, in particular, the solvability of the looping problem. Therefore, special cases are important.

2. APPLICATION TO TOPOLOGY

2.1. B-spaces

Yu.L. Ershov defined the constructivization of topological spaces with a countable basis of neighborhoods (A-spaces) [7, 8]. Let us give a generalized definition for the case of subspaces of arbitrary separable spaces with a countable basis of neighborhoods, using the ideas of P. Martin-Löf [9] and A. Lacombe [10]. In doing so, we will eliminate the initial binding to natural numbers, which is adopted by the above authors and is standard for all works on constructive topology listed in the fundamental monograph [11]. When dealing with natural numbers, we will explicitly note this fact.

Definition 13. A $\mathbf{B}_0$-space is a separable complete space in which a countable basis of neighborhoods is selected. This basis, with the empty set added, will be denoted by $\mathfrak{A}$. Let us define a base on $\mathfrak{A}$.

1. The constants $\varnothing$ and $\mathfrak{U}$ (an empty set and the entire space).

2. A general countability function $e:(\mathfrak{A} \to \mathfrak{A})$ such that the sequence $\lambda n.e^n(\varnothing)$ runs through the entire space $\mathfrak{A}$ without repetition.[4]

3. The general intersection operation of neighborhoods, $A \cap B$.

4. The general predicate $A \le B$ (A is *thinner* than B), meaning that either the closure of A is nested in B, or $A = B$ and B is an open-closed singleton.

5. The general predicate $A \# B$ (A is *separated* from B), meaning that the closures of their neighborhoods do not intersect.

6. (Optional) Some additional functions and predicates on $\mathfrak{A}$. ♦

Item 2 defines a bimorphism $\mathfrak{A}$ in **nat**, which is not an isomorphism: the base does not contain the function Pd and the equality predicate (even $a = \varnothing$). It is necessary to guarantee constructivization of the countability of $\mathfrak{A}$. According to Definition 2, n-tuples of neighborhoods are also constructive objects.

Proposition 1. *If the equality predicate of neighborhoods $a = b$ is computable in a space $\mathbf{B}_0$, then the neighborhoods form a model of natural numbers and all partially recursive functions on them are computable.*

P r o o f. To establish this result, it suffices to construct the function $Pd(a)$ since $Z(a) \equiv a = \varnothing$.

$$\begin{cases} Pda(x,\ a) & \leftarrow \textbf{if } a = \varnothing \textbf{ then } \bot \\ & \quad \textbf{elif } e(x) = a \textbf{ then } x \textbf{ else } Pda(e(x),\ a) \textbf{ fi} \\ Pd(a) & \leftarrow Pda(\varnothing,\ a). \end{cases}$$

If there is an equality, then known functions of natural numbers are freely used, in particular, tuples and operations over them. In this case, tuples are also tuples of neighborhoods. ♦

Definition 14 (a metric $\mathbf{B}_0$-space). A $\mathbf{B}_0$-space is *metric* if there exists a computable general measure function μ that assigns to each neighborhood a rational number $\mu(a) \geqslant 0$ and:

1. $\mu(A) = 0$ if and only if A contains at most one point.

2. If $A \le B$ and $A \ne B$, then $\mu(A) < \mu(B)$.

3. If $A \# B$, $A \le C$, and $B \le C$, then $\mu(A) + \mu(B) < \mu(C)$.

4. Every point $x \in \mathbf{B}_0$ has a base neighborhood of arbitrarily small measure.

Example 3. If the basic neighborhoods in a certain space have a tree of nesting, then $A \le B$ means that when B is not a leaf, A lies on a path from B. But a leaf is the end point of a path, and then $A = B$.

[4] In this case, λ is a commonly accepted quantifier of functionality and belongs to the metalanguage.

Consider Hausdorff $\mathbf{B}_0$-spaces. It is natural to consider the equivalence class of computable convergent sequences of nested neighborhoods as a computable element of a $\mathbf{B}_0$-space. According to N.A. Shanin [12], this is not sufficient: it is also necessary to have an explicitly given computable convergence regulator for such a sequence. Since no measure or uniform structure of neighborhoods is assumed, one must be careful.

Definition 15 (a computable element). A *Shanin point* is a pair of everywhere defined functions f, r, where $\forall x(f(e(x))) \le f(x))$, and the function $r(x, y)$ (*regulator*) is such that, for any pair of neighborhoods $x \le y$, either $f(r(x, y)) \le y$ or $f(r(x, y))\#x$. If f, r are computable, then an element of a $\mathbf{B}_0$-space is *constructive* relative to the given base.

Shanin points (f, r) and (f_1, r_1) are *equivalent* $(f, r) \equiv (f_1, r_1)$ if, for any n, $f(r(f_1(e(n)), f_1(n))) \le f(n)$ and $f_1(r_1(f(e(n)), f(n))) \le f_1(n)$.

Remark 4. The essential meaning is that a regulator allows getting either into a smaller neighborhood or into the gap between the smaller and larger ones. Hence, it is possible to avoid the case, unpleasant from a constructive standpoint, when the boundary of the tested neighborhood falls exactly on the limit *f* (as a result, it cannot be separated from the members $f(n)$, whereas $f(n)$ cannot get inside).

Definition 16. A **B**-space is a non-empty subset of a $\mathbf{B}_0$-space X_ϖ (the parent space), $X \subseteq X_\varpi$, with a base of generalized neighborhoods, each representing the intersection of a basis neighborhood from $\mathfrak{A}$ with the set X. ♦

Thus, all of the above functions and predicates are inherited by a **B**-space from its parent $\mathbf{B}_0$-space.

Lemma 4 (*properties of Shanin points*).

1. *If a pair of functions is a Shanin point, then the intersection of all $f(a)$ contains at most one element.*

2. *In a complete space, this intersection is non-empty.*

3. *In a metric space, a Shanin sequence f has a measure of members tending to zero.*

P r o o f. Consider two different points x, y. Then, by separability, there are basic neighborhoods $x \in A$, $y \in B$, $A\#B$. Let us take $A_1 \le A$, $x \in A_1$. Since $f(r(A_1, A)) \le A_1$, we have $f(r(A_1, A))\#B$, and thus y does not belong to the intersection of all $f(n)$. Item 1 is established. Item 2 is valid by the definition of completeness. Item 3 is satisfied because, for any neighborhood where $\bigcap_a f(a) \in A$, we have $f(r(A)) \le A$. ♦

In the case of metric spaces (see Martin-Löf's and other works listed in [13]), regulators are not needed, as it suffices to require rapid convergence of a sequence (e.g., that the measure of each $f(n)$ does not exceed 2^{-n}). In the general case, in the absence of regulators, the counterexamples from the fundamental work [12] are valid, destroying the constructiveness of considerations.

Proposition 2. *Any separable space Ξ with a countable base of neighborhoods can be represented as a* **B**-*space.*

P r o o f. A $\mathbf{B}_0$-space Ξ_0 is the complement of Ξ. It also has a countable base of neighborhoods. We define a base on it and obtain a **B**-space. The neighborhoods will be the intersections of the neighborhoods from the base Ξ_0 with Ξ.

Note that the base for Ξ is not specified. All operators work over Ξ_0 and the belonging to Ξ is specified externally and is not used in any computations.

Definition 17. A *computable function* over the elements of **B**-spaces, $f : \Xi \to \Upsilon$, is a computable operator

$$\Phi : (((\mathfrak{A}, \mathfrak{A}) \to (\mathfrak{A}, \mathfrak{A})), \mathbf{nat}, \mathbf{nat} \to (\mathfrak{A}, \mathfrak{A}))$$

that transforms any Shanin point (f, r); $\bigcap_a f(a) \in \Xi$ into a function $\lambda n, m.\Phi((s, r), n, m)$ that is a Shanin point (g, q); $\bigcap_a g(a) \in \Upsilon$, where

$$(s, r) \equiv (s_1, r_1) \supset \Phi((s, r)) \equiv \Phi((s_1, r_1)). \quad (3)$$

Here, (s, r) is understood in the sense of the transformation of functions defined above.

Remark 5. Thus, Definition 17 can be translated into a more familiar language: for a Shanin pair (s, r) representing x, Φ yields a Shanin point (g, q) representing $f(x)$ as follows:

$$\Phi(s, r, i, j) = (g(i), q(i, j)).$$

The arguments of the functions run through the entire space Ξ_0; only the results and the correctness requirement are restricted. Totality and generalness are also considered on the entire space Ξ_0.

Example 4. This example illustrates the important role of continuity regulators. Let us take the common space of real numbers with a basis of neighborhoods defined by the intervals of rational points $\left(a - \frac{1}{2^n}, a + \frac{1}{2^n}\right)$. Consider an operator that processes each neighborhood $\left(a - \frac{1}{2^n}, a + \frac{1}{2^n}\right)$ into $\left(1 - \frac{1}{2^n}, 1 + \frac{1}{2^n}\right)$ if its lower bound is greater than zero, into $\left(-1 - \frac{1}{2^n}, -1 + \frac{1}{2^n}\right)$ if the upper bound is less than zero, and into $\left(-\frac{1}{2^n}, +\frac{1}{2^n}\right)$ if 0 is inside the interval. It processes every computable convergent sequence into a computable convergent sequence, but a regulator cannot be obtained because it must yield neighborhoods of zero for sequences whose members all include zero. But then it must yield $\left(-\frac{1}{2}, \frac{1}{2}\right)$ for a finite number of its members, and by replacing it with a sequence that separates from zero at the next step, we obtain an incorrect result (the deception method).

Remark 6. Functions are not assumed to be defined everywhere as operators on elements of a space. Moreover, there arises another case when a function is not defined, i.e., if the correctness condition (3) fails for Shanin pairs representing a given element. Correct operation must be ensured only on elements of the set Ξ. The equivalence of result regulators for different Shanin representations of the argument is not required.

2.2. The Main Theorem

Theorem 4. *A function is continuous on a* **B**-*space if and only if it is computable relative to some base.*

The proof of this theorem requires several additional constructs.

Lemma 5. *A computable function transforms constructive points of Ξ into constructive points of Υ.*

Corollary of Lemma 2. Note that the arguments of a computable function are not assumed to be computable. It must work correctly on any arguments. This is a fundamental difference from standard concepts of constructiveness [11], even if they are relativized.

Lemma 6. *All computable functions are continuous on their definitional domain.*

P r o o f.

Corollary of Theorem 1 (on continuity). Let Y be a neighborhood of the result g, q. Then its regulator gives $q(Y)$ such that $g(q(Y)) \le Y$. According to the continuity theorem, only a finite number of values $[f(\varnothing), \ldots, f(e^n(\varnothing))]$ of the argument function f are used to find $g(q(Y))$. Now let us take an arbitrary Shanin point f_1, g_1 from $f(e^n(\varnothing))$. By replacing $[f_1(\varnothing), \ldots, f_1(e^n(\varnothing))]$ with $[f(\varnothing), \ldots, f(e^n(\varnothing))]$ and all values $g_1(a)$ such that $f(e^n(\varnothing)) \le g(g_1(a))$ with $e^n(\varnothing)$, we obtain an equivalent Shanin point. It belongs to $g(q(Y))$ and, hence, to Y. Thus, for any neighborhood of the result, we can find a neighborhood of the argument that maps into it.

The main theorem is proven in one direction. For each continuous function on a **B**-space, it remains to find a base relative to which this function will be constructive. ♦

Functions in this context are not full-fledged values. There is no collection of computable functions. They form a set that is external to the computational model. This is especially true for operators. However, there is a well-known topological transformation, i.e., one can move from a continuous function to a relation between the neighborhoods of the result and the argument. Its constructive form was proposed by Martin-Löf [9]: here, it will be generalized to non-metric spaces.

Definition 18 (type demotion). An *approximation* is a function $\mathbb{A}$ enumerating neighborhoods such that, for any a and b, there exists $c: \mathbb{A}(a) \cap \mathbb{A}(b) = \mathbb{A}(c)$. An approximation is *maximal* if for any $a \le b$ there exists c such that $a \# \mathbb{A}(c)$ or $\mathbb{A}(c) \le b$.

An open set is a function $\mathbb{O}$ whose values are neighborhoods and, for any a, if $b \le \mathbb{O}(a)$, then there exists c such that $\mathbb{O}(c) = b$.

A *neighborhood relation* is a function R from $\mathfrak{A}$ into pairs of neighborhoods (X, Y) of points $x \in \Xi, y \in \Upsilon$ such that $R < X >$ is an approximation in Υ and $R^{-1} < Y >$ is an open set in Ξ. ♦

To ensure constructiveness, we design enumerating functions of an approximation and open sets via R.

Lemma 7 (*a function as a relation*). *For any continuous function $f : \Xi \to \Upsilon$ of* **B**-*spaces, one can find a neighborhood relation R such that for any neighborhood a of the argument x, $R < a >$ contains $f(x)$, for a neighborhood b of the result $f(x)$, the union of neighborhoods $R^{-1} < b >$ contains*[5] *$f^{-1}(f(x))$, and for any $z \# f(x)$, there exists a neighborhood a such that some neighborhood from $R < a >$ is separated from z.*

P r o o f. Let us take an arbitrary continuous function $f : \Xi \to \Upsilon$. Since f is continuous, for any neighborhood Y of the result $f(x)$ there exist neighborhoods X of the argument x such that $f(x) \in Y$ for any $x \in X$. They form the desired relation R. ♦

At the same time, $R < X >$ is maximal for $x \in X$. Let us construct a search function *sf*(a,b) that is computable relative to the parent space and R:

```
sf0(a, b, x) ← if $a \le b$ then
                   if $R(a, x)$ then
               if $x \le a$ then $x$ elif $x \# a$ then $x$ fi          (4)
                   else $sf0(a, b, x+1)$ fi
               else $\perp$ fi
sf(a, b)     ← $sf0(a, b, 0)$.
```

[5] f^{-1} is treated as a relation, not as a function: $R^{-1} = \{(x, y) \mid (y, x) \in R\}$.

This function finds, in approximation, a neighborhood that is either nested in or separated from the smaller of any pair of nested neighborhoods from the approximation.[6]

Lemma 8 (type promotion). *If a neighborhood relation R represents a function f, then f is computable relative to the parent* **B**$_0$*-spaces, the predicate $=$, and R.*

P r o o f. Since there is the equality predicate, neighborhoods can be identified with natural numbers, and standard recursive functions can be used accordingly.

Let us construct the result sequence $\lambda n.g(n)$ using a function f from a Shanin point. The invariant of the function created is:

For any n, $f(n), g(n) \in R$ and $g(n+1) \le g(n)$.

We define the auxiliary function $sr(a, b)$:

$$\begin{cases} sr0(a, b, n) & \leftarrow \textbf{if } pr_1(R(n)) = a \wedge pr_2(R(n)) \le b \\ & \quad \textbf{then } pr_2(R(n)) \textbf{ else } sr0(a, b, S(n)) \textbf{ fi} \\ sr(a, b) & \leftarrow sr0(a, b, 0). \end{cases}$$

Provided that the computation of sr for given a,b is finite, it satisfies the following invariant:

$$(a, sr(a, b)) \in R \wedge sr(a, b) \le b.$$

The function g is defined by the schema

$$g(n) \leftarrow \textbf{if } Z(n) \textbf{ then } f_2(0) \textbf{ else } sr(f_1(n), g(Pd(n)) \textbf{ fi}.$$

The finite computation of g is guaranteed if f belongs to the definitional domain of the function serving as the base for constructing R.

The regulator for the neighborhood relation has already been constructed above (see formula (4)); we only need to replace the found neighborhood with its number in the result of the desired function.

Thus, the main theorem is proved. ♦

It has the following corollary.

Lemma 9. *The result f(x) of a continuous function is constructive relative to R and x.*

P r o o f. The function sf gives the result, and sr gives its regulator.

2.3. Computability of the Neighborhood Relation

The question arises about the computability of the neighborhood relation relative to the base for which a given continuous function is computable. Here, one has to use multistage nontrivial transformations of the function definition and a nontrivial programming technique. Since this technique is secondary to the logical results of the paper, here we will describe only the ideas for such transformations and formulate the results; the study of the resulting concept of computability, which has some nontrivial properties and, at the same time, sufficient power, is the subject of other research works.

To compute the neighborhood relation, we would like to run the function at least for all constructive numbers. This is impossible due to the lack of means to change the values of functional variables. However, this difficulty can be circumvented for general functions by using the main theorem. Therefore, the proof of the theorem requires several auxiliary constructs. Being of independent interest, they are carried out for the general case, but involve quite cumbersome technicalities. The idea behind them is simple and transparent.

Theorem 5 (computation of the relation). *By the definition of an operator Φ that computes a general function f, it is possible to construct a definition of the neighborhood relation for this function in the same base supplemented by the predicate of equality.*

P r o o f. After an analysis of the function's definition, we construct an enumeration of the generalized graph of the operator (the main technical part of the work) and replace in it all pairs $((F, y), z)$ with $(x, pr_1(z))$, where x is the smallest neighborhood in $pr_1(F)$. The first element of the Shanin pair decreases monotonically; hence, if there are enough other elements surrounding it, the function's result will fall within a neighborhood from $pr_1(z)$ in the neighborhood of x, and the constructs of the regulator do not affect the function's value and are omitted.

2.4. Relative Constructiveness

The question arises: is relative constructiveness a characteristic of some class of functions?

Definition 19 (relative constructiveness). A function ψ on a **B**-space is *constructive relative to* $f:(\mathbb{N} \to \mathbb{N})$ if, for every number x, there exists a constructive function φ over the base S, Pd, Z, f such that $\psi(x) = \varphi(x)$. ♦

First of all, the following answer is trivial: this class is broader than computable (continuous) functions. The result of the Dirichlet function is 0 or 1 and is constructive. But this function itself is not constructive relative to any basis.

Definition 20. A function is *countably continuous* if the **B**-space can be divided into a countable set of **B**-spaces on each of which it is continuous.

Theorem 6 (the equivalence of concepts). *A function is relatively constructive if and only if it is countably continuous.*

[6] From a constructive standpoint, proving the correctness of this function needs application of Brouwer's bar induction [16], which once again demonstrates that the countability of the basis of neighborhoods is an important requirement.

P r o o f.

The necessity part. Let a function be relatively constructive. Since each of its results is generated by an operator defined over the base S, Pd, Z, F and is a constructive function, we can assign to all numbers the operators that compute them. Since the collection of computable operators is countable and each constructive function is continuous, we arrive at the required decomposition.

The sufficiency part. Let a function be countably continuous. We take the neighborhood operators Φ_i defining each of the continuous fragments and combine them into the function $\Psi(i, n) = \Phi_i(n)$. Then each $f(x)$ is computable relative to Ψ. ♦

This design is fundamentally nonconstructive. It is impossible to combine constructive fragments into a uniform constructive function on the space. Thus, the question of which function applies to a given element is unsolvable relative to any base. Note that the above constructs do not involve the predicate of belonging of an element to the **B**-space support or the properties distinguishing the **B**-space from the parent space.

Example 5. Finally, we construct a function of a real variable that is not countably continuous. Let us take the first ordinal of the cardinality of the continuum and order the real numbers according to this ordinal as follows. To each ordinal, we assign a real number that cannot be constructively represented via the previously ordered numbers. Such a number can be found since the cardinality of each ordinal number smaller than the first continuum one is less than the continuum, and only a countable collection of numbers can be constructively defined via each number. Then the function assigning to each x_α its follower $x_{\alpha\oplus 1}$ is not relatively constructive for any number and, therefore, is not countably continuous.

3. DISCUSSION AND APPLICATION OF RESULTS

3.1. The Relationship with Other Concepts of Constructiveness in Topology

First of all, we note that in the works used and described in [11], the continuity of constructive operators, in one form or another, was assumed in advance.

The continuity of functions of a real variable was proven completely independently only in soviet constructivism [14], albeit for "hackerish" operators. First, such operators have to process only constructive functions; second, the source code of the algorithm of any function is considered known, and anything can be done with it. As shown in [15], the first feature is harmless when discarding the second: the space of functions in intuitionism can consist only of algorithmically computable ones, but there should be no access to their programs.

A slight modification of the definition of higher-type operators (the absence of initial, externally given higher-order operators) led to strong Baire continuity and, simultaneously, to the possibility of processing any functions. This continuity can be justified intuitionistically by accepting Brower's bar induction principle [16]. Thus, another step was taken. One can assume the existence of function algorithms when allowing to use them as in modern computer science: by calling closed modules.

In topology, we adhere to the lines of Ershov [7, 8] and Martin-Löf [9]. For instance, Martin-Löf used algorithms and introduced topology as point-free, based on approximations, and functions on topological spaces as neighborhood relations. Martin-Löf's followers limited their consideration to compactness and, as a consequence, traditional spaces of real numbers as well as Cantor and Baire spaces; they were more inclined towards formal topology. The research works of Martin-Löf's line were reviewed in [17].

E. Bishop developed a concept of constructiveness, which was aptly characterized in a conversation by A. G. Dragalin: use only algorithms, but never confirm or deny this [18]. Bishop's concept was developed [19–21] by introducing the notion of continuity indirectly, through the Heine–Borel theorem, which is equivalent to Brouwer's bar induction. This line of research, based on Bishop's concept, was reviewed in [13]. The notion of a subspace in both schools is subject to strong restrictions.

The notion used here is more abstract and independent of a particular basis of computability. With this notion, we can consider arbitrary subspaces selected in a nonconstructive way, as well as avoid the use of a distinguishing predicate in constructive design. In addition, the derivative concept of a base on a **B**-space does not satisfy the conventional requirements for a base of open sets. In particular, objects intersecting as basic neighborhoods may have empty intersection as sets, and basic neighborhoods as sets may be empty. However, they are inherited from the parent $\mathbf{B}_0$-space, allowing one to handle them correctly.

We have succeeded in combining the advantages of the approximation approach (from the author's standpoint, equivalent to the formal topology approach) and the pointwise approach. Also, we have succeeded in defining functions purely functionally (through elements) and obtaining both continuity and the capability to process elements not specified by algorithms. A partial similarity with soviet constructivism is the requirement that operators are specified by a program; but this program is used correctly as a callable module only to conduct experiments, being applied to various arguments (including those specified externally), and the collection of programs is not used as a whole.

The only significant limitation remaining is the countable basis of open sets. Additional studies are needed to find out the transferability of the results to inseparable spaces and spaces with an uncountable base.

3.2. Connection with Applications and Metrology

Hereinafter, *computational problems* are understood as those related to real numbers or other Hausdorff spaces, as opposed to discrete problems.

When defining a regulator, the subtlety corresponds to the absence of the dichotomy $\forall x, y(x \le y \lor y \ge x)$ in constructive analysis and its replacement with the inaccurate comparison

$$\forall \varepsilon(\varepsilon > 0 \supset \forall x, y(y > x - \varepsilon \lor y < x + \varepsilon)).$$

In computational practice with the representations of real numbers with overlap, this makes the equality operation difficult to compute, and real numbers are rejected on this base. From a physical viewpoint, however, *there are no exact real numbers*. In algorithms, comparing numbers for equality leads to a multitude of difficult-to-detect errors and instabilities. In each case, it would be necessary to consider the degree of precision required to compare data, but this is hindered by a pedagogical issue: almost everywhere, rational numbers are treated as a subset of real numbers, which ignores their fundamentally different nature.

Accordingly, we arrive at another practically important conclusion: the frequently encountered objection to represent real numbers by overlapping systems (see an example of such systems in [22])—the complexity of computing the equality—is unreasonable from applied and scientific standpoints.

The equivalence between constructiveness and the purely topological concept established in this paper is new. A distant analogy is the following theorem: the cardinality of a collection of continuous functions on a space with a countable base of neighborhoods is the continuum. Previously, constructiveness was only a special case of topology, but now it turns out to be topology itself. This is a consequence of an important methodological principle: *any binding to particular representations and data structures limits our view*. (Recall I. Kant, who said that we can only think of things in space and time; this is unfair to modern logic, mathematics, and physics.) And real numbers most often turn out to be such structures. Therefore, the transition to the most abstract representations has repeatedly demonstrated its power in mathematics and logic. However, one should keep in mind that in this case, the transition from ideal objects to real ones is more difficult and may involve multiple stages.

The initial "undeveloped" concepts of Martin-Löf and Shanin have proven to be the best for generalization and modification. This confirms another methodological principle: *optimization reduces flexibility, i.e., the capability to generalize and change*. In evolution, this means the extinction of well-adapted species when the environment changes.

Martin-Löf was the first to show, using examples, that the same function spaces in constructive mathematics can have different spaces of constructive operators. But he did not explicitly emphasize the difference between his operators and those of soviet constructivism. Later, it became clear that discrepancies could also arise at higher levels. The concept of computability proposed in this paper is one example. It differs from Martin-Löf's concept at the third level.

From a practical viewpoint, we can draw the following conclusion: if a program is written in a functional style without using incorrect operations over real numbers from a constructive standpoint (equality, $\ge, \operatorname{sign}(x)$, etc.), then it defines a continuous function, and no other proofs are needed. In addition, the restrictions accepted on higher-order computability show why the style of functional programming in numerical problems is different from that in discrete counterparts: for example, categorical constructs are not used. Categorical operators are of a higher order than those allowed in the base, and they destroy computability on topological spaces (see Example 1). But their use as macros deteriorates nothing.

A theoretical and practical question arises: what program transformations are allowable as macros in computational problems? It seems that this is supercompilation [23]. But it has been studied primarily for discrete problems so far.

CONCLUSIONS

This study raises a series of theoretical questions concerning the concept of computability used. It is formally weaker than the common concept of computable operators of finite type. While eliminating the need to explicitly construct models of sets of operators of finite types, it serves to solve fairly strong problems. Further research is required here, particularly into possible extensions preserving topological properties and allowing the computation of operators with external program restructuring required.

The results of subsection 2.4 were partially presented at the Thirteenth National Supercomputing Forum (NSCF-2024) and published in its online proceedings. The results of this work were announced at the Smirnov Readings. The full text of the paper was reported at Ailamazyan Program Systems Institute, the Russian Academy of Sciences, on April 10, 2025.

Acknowledgments. *This work was carried out within project no. 125021302067-9 of Ailamazyan Program Systems Institute, the Russian Academy of Sciences.*

REFERENCES

1. Mitchell, J.C., *Foundations for Programming Languages*, Cambridge: MIT Press, 1996.
2. McCarthy, J., A Basis for a Mathematical Theory of Computation, in *Computer Programming and Formal Systems*, Braffort, P., and Hirshberg, D., Amsterdam: North-Holland, 1963, pp. 33–70.
3. Barendregt, H.P., *The Lambda Calculus. Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, Amsterdam: North-Holland, 1984.
4. Mitchell, J.C., *Concepts in Programming Languages*, Cambridge: Cambridge University Press, 2003.
5. Stark, R.W., *LISP, Lore, and Logic*, New York: Springer-Verlag, 1990.
6. Strachey, C. and Wadsworth, C.P., Continuations: A Mathematical Semantics for Handling Full Jumps, *Higher-Order and Symbolic Computation*, 2000, vol. 13, no. 1/2, pp. 135–152.
7. Ershov, Yu.L., Theory of *A*-spaces, *Algebra i Logika*, 1973, vol. 12, no. 4, pp. 369–416. (In Russian.)
8. Ershov, Yu.L., *Teoriya numeratsii* (Theory of Numberings), Moscow: Nauka, 1977. (In Russian.)
9. Martin-Löf, P., *Notes on Constructive Mathematics*, Stockholm: Almqvist & Wiksell, 1970.
10. Lacombe, A., Quelques procedes de definition en topologie recursive, in *Constructivity in Mathematics*, Amsterdam: North-Holland, 1959, pp. 129–158.
11. Bridges, D., Ishihara, H., Rathjen, M., and Schwichtenberg, H., *Handbook of Constructive Mathematics*, Cambridge: Cambridge University Press, 2023.
12. Shanin, N.A., Constructive Real Numbers and Constructive Functional Spaces, *Trudy Mat. Inst. Steklov.*, 1962, vol. 67, pp. 15–294. (In Russian.)
13. Kawai, T., Bishop Metric Spaces in Formal Topology, in *Handbook of Constructive Mathematics*, Bridges, D., Ishihara, H., Rathjen, M., and Schwichtenberg, H., Eds., Cambridge: Cambridge University Press, 2023, pp. 395–425.
14. Kushner, B.A., *Lectures on Constructive Mathematical Analysis*, American Mathematical Society, 1984.
15. Kreisel, G. and Troelstra, A.S., Formal Systems for Some Branches of Intuitionistic Analysis, *Ann. Math. Logic*, 1970, vol. 1, no. 3, pp. 229–387.
16. Heyting, A., *Intuitionism: An Introduction*, Amsterdam: North-Holland, 1966.
17. Ciraulo, F., Subspaces in Pointfree Topology: Towards a New Approach to Measure Theory, in *Handbook of Constructive Mathematics*, Bridges, D., Ishihara, H., Rathjen, M., and Schwichtenberg, H., Eds., Cambridge: Cambridge University Press, 2023, pp. 426–444.
18. Bishop, E., *Foundations of Constructive Analysis*, New York: McGrawHill, 1967.
19. Kawai, T., Localic Completion of Uniform Spaces, *Log. Meth. Comput. Sci*., 2017, vol. 13, no. 3, art. no. 22, pp. 1–39.
20. Kawai, T., Point-Free Characterisation of Bishop Compact Metric Spaces, *J. Log. Anal*., 2017, vol. 9, no. 5, pp. 1–30.
21. Kawai, T., A Point-Free Characterisation of Bishop Locally Compact Metric Spaces, *J. Log. Anal*., 2017, vol. 9, no. c2, pp. 1–41.
22. Nepeivoda, N.N., Grigorevsky, I.N., and Lilitko, E.P., New Representation of Real Numbers, *Program Systems: Theory and Applications*, 2014, vol. 5, no. 4 (22), pp. 105–121. URL http://psta.psiras.ru/read/psta2014_4_105-121.pdf. (In Russian.)
23. Klimov, A.V. and Romanenko, S.A., Supercompilation: Main Principles and Basic Concepts, *Preprints of the Keldysh Institute of Applied Mathematics*, Moscow, 2018, no. 111. DOI: 10.20948/prepr-2018-111. URL: http://library.keldysh.ru/preprint.asp?id=2018-111. (In Russian.)

This paper was recommended for publication
by RAS Academician S.N. Vassilyev,
a member of the Advisory Board.

Received May 13, 2025,
and revised August 25, 2025.
Accepted October 2, 2025.

Author information

Nepeivoda, Nikolai Nikolaevich. Dr. Sci. (Phys.–Math.), Ailamazyan Program Systems Institute, Russian Academy of Sciences, Pereslavl-Zalessky, Russia
✉ nnn@nnn.botik.ru
ORCID iD: https://orcid.org/0000-0001-8845-7627

Cite this paper

Nepeivoda, N.N., Continuity as Computability. *Control Sciences* 6, 20–32 (2025).

Translated into English by *Alexander Yu. Mazurov*,
Cand. Sci. (Phys.–Math.),
Trapeznikov Institute of Control Sciences,
Russian Academy of Sciences, Moscow, Russia
✉ alexander.mazurov08@gmail.com