The Role of Functional Programming in the Organization of Parallel Computing

Lidia V. Gorodnyaya

A.P. Ershov Institute of Informatics Systems (IIS), SB RAS, Acad. Lavrentjev pr., 6, Novosibirsk 630090, Russia

Abstract

The article is devoted to the results of the analysis of modern trends in the field of functional programming, considered as a methodology for solving problems of organizing parallel computing. The paradigm analysis of languages and functional programming systems is involved. Taking into account paradigmatic features is useful in predicting the course of application processes of programs, as well as in planning their study and development. Functional programming helps to improve the performance of programs by preparing their prototypes in advance. The description of the semantic and pragmatic principles of functional programming and the consequences of these principles is given. The complexity of creating programs for solving new problems is noted. The role of the paradigmatic decomposition of programs in the technology of developing long-lived programs is noted. The perspective of functional programming as a universal technique for solving complex problems, burdened with difficult to verify and poorly compatible requirements, is especially emphasized. Paradigm analysis of programming languages and systems allows to reduce the complexity of the problems being solved by methods of decomposition of programs into autonomously developed components and prototypes, which also reduces the labor costs of developing programs. A variety of paradigmatic characteristics inherent in the preparation and debugging of parallel computing programs is shown.

Keywords

Functional programming, parallel computing, programming languages, programming paradigm, multi-paradigm

1. Introduction

Functional programming is one of the first paradigms aimed not only at obtaining an effective implementation of prepared algorithms, but at solving new information processing problems with a research component [1–8]. If the number of programming languages is now growing rapidly, then the number of paradigms is not so great. This situation allows for a fairly complete comparison and selection of characteristics for a clear separation of paradigms and understanding the reasons for their diversity. Considering the results of the analysis of tasks, means and methods of organizing parallel computations, one can draw attention to the obvious variety of problem settings and corresponding priorities in decision-making at different stages of program development and debugging. The programming language comparison methodology presented in [9–12] is based on the informal definition of the term “programming paradigm” given in [13], which states that when comparing paradigms one should highlight the distinctive testable features. As such signs, it turned out to be useful to use priorities when making decisions at different stages of program development and debugging. As a result, the methodology of paradigmatic analysis makes it possible to reduce the formulations of the tasks being solved to sets of autonomously developed subtasks, to assess their similarities and differences. It is useful to take this into account when predicting the complexity of the processes of applying programmable solutions, starting with planning, studying and organizing the development of long-lived programs [14, 15].

SSI-2021: Scientific Services & Internet, September 20–23, 2021, Moscow (online)
EMAIL: lidvas@gmail.com (L.V. Gorodnyaya)
ORCID: 0000-0002-4639-9032 (L.V. Gorodnyaya)

The telling begins by looking at a number of parallel computing paradigms supported in well-known programming languages. Then the basic principles of functional programming and their concretization arising when transferring functional programming methods to parallel computations are discussed. A number of parallel computing paradigms supported in well-known programming languages are considered. Then the problem of taking into account the degree of study of parallel computing tasks and the peculiarities of the transition to production functional programming is analyzed. Separately, an assessment of the complexity of parallel programming is considered, taking into account the degree of study of parallel computing problems and the level of qualifications of specialists graduated from the educational system. In the conclusion, the requirements for parallel programming language are described.

2. Paradigmatic characteristic

Considering the ideas of functional programming of parallel computations, it should be noted that a significant number of parallel programming languages have already been created. It is possible that one of the reasons for the complexity of developing programs for parallel computing is their hidden multiplicity of paradigms [16–18]. To develop a program for parallel computing, a lot must be thought out in different ways simultaneously in different paradigms, convenient for solving individual subtasks without the possibility of solving a full complex of subtasks in a common environment. Obviously, the tasks of scaling computations, synchronizing interactions in multithreaded programs, representing natural asynchronous parallelism and achieving high program performance are different. A separate task is to familiarize oneself with the phenomena of parallelism in order to give specialists an intuitive idea of methods for solving problems that depend on unusual phenomena. New works on functional programming is essentially aimed at finding more efficient solutions to parallel programming problems.

2.1. Paradigms of parallel programming languages

A noticeable number of programming languages and systems have been created that allow solving some of these problems within the framework of specific paradigms, the support of which is presented in different programming languages. (see Table 1).

Table 1
The parallel computing problems and paradigms supporting in different programming languages and tools

Problems	Paradigms	Programming languages and tools
Scaling	**Multiprocessor** programming	VHDL, XC, SIGMA, bash, Occam, mpC, EL'-76, Limbo, Kotlin, MPI
Thread synchronization	**Sync** programming	APL, VAL, Sisal, Alef, E, X10, LuNA, Charm, Go, Java, Scala, Rust, Pifagor, OpenMP
Problems statement	**Asynchronous** programming	BARS, Haskell, Erlang, JavaScript, Python, C#
Programs performance	**High performanc**e programming	Setl, HPF, G, Sparkel, mpC, Sanscript, D, Rest, F#
Concurrency familiarization	**Instructional** programming	Logo, Робик, Karel, OZ, A++, Sinkhro, Lsl

Thus, for each of the hard-to-solve problems of parallel computing, a separate convenient paradigm for its solution has already been formed and a number of programming languages have been created that support this paradigm. The difference between paradigms is manifested in the ordering of the importance of the means and methods used in solving individual problems, other tasks require a different ordering. One paradigm is usually used at each point in the development of a program. Accordingly, there is one leading paradigm in each programming language. The requirements for solving rather complex problems of parallel computing are associated with a number of difficult problems, which entails the need to use different paradigms at separate stages of their creation and phases of their life. During the transition to programming

technology, it is important to ensure that practical results are obtained, which requires the support of the full range of paradigms used at different stages of program development that form its life cycle. The complexity of using different paradigms in solving one problem is usually minimized by creating multilingual systems that allow, as necessary, the possibility of transition from one paradigm to another without the cost of mastering different interfaces and systems. This leads to the expediency of creating a multi-paradigm parallel computing language that simultaneously supports all the main parallelism paradigms.

As the experience of the BARS and Haskell languages shows, in such cases it is convenient to decompose the programming language definition into separate sublanguages that support the main paradigms or monads aimed at specific models for preparing and presenting programs. It is important that at each stage of program development the use of one paradigm is localized, characterized by a relatively small set of tools and methods within one way of thinking. Each paradigm has its own content for the categories of semantic systems and its own ordering of their role in the programming process [10, 18].

Multiprocessor programming tools usually rely on data structures and characteristics of the available architecture, including the underlying multiprocessor complex and the relationships between its elements. Operating the complex allows you to initiate the processes of functioning of individual processors, their blocking, resumption and cancellation of processes. In the course of the processes, data exchanges are possible according to certain protocols, the final results of which can be considered as the goal of the program. Decision making begins with defining the space of possible multiprocessor complexes, which can be considered as a special kind of memory with its own discipline of functioning and interaction of elements. Next comes the selection of suitable configurations and the appropriate structuring of the iteration space of the processes intended to be executed on individual processors. Then the resulting process control scheme is filled with the actual actions that perform the calculations. Routines without recursion are generally preferred, freed from other computational management complexities and hard-to-predict side effects of shared memory. A multiprocessor program is being built that allows you to dynamically reconfigure a multiprocessor complex [2].

In multi-threaded programming, sufficiently clear control schemes for computations are distinguished and regular sections and typical program models that are convenient for parallelization are separated. Usually, fragments free from side effects in memory are allocated, and non-imperative control of the execution time of program threads is allowed, taking into account the hierarchy of the program control scheme and some timing relations. Decision making begins with the choice of standard control schemes, the concept of “iteration space” is used, which can be structured depending on the distribution of data and methods of storage, which can affect the efficiency and discipline of processing multi-level memory. Iteration control can use arbitrary predicates. The control scheme over the stream iteration space is filled with fragments that are relatively easy to debug, possibly debugged in advance or programmed autonomously. To return the results of the program, special means are allocated for generating the results of calculations obtained on single-level threads of a multithreaded program. Thus, a multithreaded program is obtained with a dynamically variable space of threads over local memory with the possibility of episodic synchronization of their individual fragments [3].

Asynchronous programming is aimed at representing independent program elements that reflect the nature of the problem being solved, which can be the basis for maximum parallelization, provided that special schemes for organizing computations are identified, taking into account the characteristics of the available equipment. Decision-making begins with the choice of action control schemes and the definition of the conditions for their start. Actions can use one or another discipline of processing different types of memory. An implicit and programmable variety of memory access disciplines are supported, including a heterogeneous memory hierarchy, and computation schemes designed to include fragments that are procedures or library units. The result is a program, which is a synchronizing network diagram, that asynchronously controls the execution of actions based on the conditions that they are ready for execution.

High-performance programming requires a transition from a single program run to taking into account the prospects for its repeated use and improvement. This makes it possible to use the underused capacities of the multiprocessor complex, executing separate fragments of the program for the upcoming calculations, taking into account the data that may be required in future runs of the program. Decision making begins with computation schemes and models, perhaps over shared memory, but with local memory priority. Program control takes into account the possibility of re-execution of the program when debugging and

2https://mooc.tsu.ru/mooc-openedu/mpi/ “Course Parallel programming using OpenMP and MPI”
3https://habr.com/ru/post/121925/ Course “MPI Basics”

applying it, including the benefits of inheriting results between its runs and comparing the measurable performance characteristics of versions of the program. This leads to the appearance of a number of improved versions of the program, the choice of one of which may better take into account the conditions of use, including the configuration of the multiprocessor complex and the requirements for the quality of the program.

Educational programming is designed to fill the gaps in recommendations for imperative solutions to any problems. Such reference points make it difficult to study programming methods in general, even more so for parallel computing. Learning programming can give rise to all paradigms of parallel computing, using experience in the development of game programs such as robot visualization. This is a sufficient reason for the multi-paradigm nature of educational programming languages, which usually support some means of representing parallel computations [19, 20].

Long-lived programming languages, like the new programming languages of our century, are usually multi-paradigm. Successful parallel programming practice requires support for the full range of parallel computing paradigms. It is necessary to allow their development, replenishment and application as necessary, ensuring the transition to the next paradigm without changing the programming languages and the system environment.

2.2. Labor intensity and novelty

In practical programming jargon, the term “programming language” is used as “the input language of a programming system based on a typical hardware configuration” The difference lies in the fact that the programming system usually accompanies the implementation of the programming language with an extensible set of library modules and may not support certain complexities of the semantics of the programming language. As a result, the differences that are noticeable in practice between different languages and programming systems are smoothed out. In addition, direct measurements of the complexity of programming and program performance almost do not reflect the dependence of the result on the decisions made by the programmer and the choice of programming language constructs. Although programmable solutions are presented in terms of a programming language, their influence dissolves in a very complex complex that inherits the performance of the programming system and equipment with a large dominance of the characteristics of the element base and equipment. Thus, the problem arises of creating a methodology that makes it possible to identify such dependencies by comparing the results of direct measurements with expert assessments of the features of languages and programming systems [21–23].

The dependence of the labor intensity and quality of programming on the qualification level of programmers is studied from the first steps of the formation of programming as a profession. According to J. Weinberg, the author of the unique monograph “The Psychology of Programming” [23], in the early 1970s, IBM conducted research on the spread of programming labor intensity depending on experience, age, and abilities. It turned out that for simple problems the spread is 1 to 28, and almost regardless of experience and age. On complex tasks, a positive dependence on ability, experience and age is noticeable with a slightly smaller spread, about 1 to 10. In addition, the speed of completing tasks rarely contributes to the quality of solutions; more often, on the contrary, hasty software decisions lead to great labor costs during debugging, and even more so in working with programs. No less scatter was noticed in the assessment of the quality of programmable solutions; the dependence of the performance of programs on the labor intensity of programming was not revealed. This conclusion is supplemented by the studies of V.L. Averbukh of the dependence of labor productivity in the field of high performance computing on the age qualification. The conclusion was made in favor of the older age category, who is able not only to use their experience, but also to critically perceive fashionable novelties, to find practical compositions of classics and modernity [24].

When predicting labor costs for parallel programming, one should take into account not only the complexity of the problems being solved, but also the level of study of the formulations of these tasks, and the level of qualifications and capabilities of program developers. It is important to overcome the conceptual complexity of the implemented and used programmable and software tools, the functioning of which may go beyond the usual notions. For parallel computing, the degree of knowledge often inherits the results of a previously created sequential program for solving a problem with a mathematically precise formulation. There is a temptation that inhibits awareness or the creation of a more efficient parallel algorithm. In addition, the level of qualifications of specialists is based on the experience of imperative-procedural

programming, which prevents the perception of more complex dependencies in parallel processes. The conceptual complexity of concurrent computing problems goes beyond ordinary education, and many concepts acquire a broader interpretation, in part contrary to ordinary intuitive understanding. The presentation of the results of evaluating conceptual complexity, which allows structuring the space of such parameters, is considered in articles [9–11].

The success of functional programming in practice essentially depends on the choice of problem formulations, the solutions of which are represented by systems of functions that are relatively simple, not too time consuming and convenient for debugging. In terms of the degree of study, the following categories of problem statements differ significantly, affecting the choice of methods for solving problems and the complexity of their programming:

- new,
- scientific,
- practical,
- well-posed.

The leading criterion for new and scientific problems is correctness and complete solutions. It is these criteria that functional programming is aimed at, which is the main testing ground for research and development of methods for program verification. Practical and well-posed tasks are more focused on efficiency and usability of programs. This is where the benefits of imperious and object-oriented programming come in.

The new problem statements are characterized by the absence of an accessible precedent for the practical solution of the problem, the novelty of the means used, or the lack of experience of the performers. The problems of parallel computations were posed back in the pre-computer era, and the formulations of many such problems have mathematical precision [25, 26]. Nevertheless, some of the problems of parallel programming of their solutions have to be considered as new due to the rapid updating of IT, the element base and unsolved educational problems of programming in general. Any problem that has not received a good practical solution remains in the status of a new problem, regardless of the time it was posed.

Scientific formulations of parallel programming problems currently follow the trend of the functional programming and the study of schemes for structuring the iterative space of processes. The functional programming can be viewed as a technique for reducing solutions of complex problems to compositions from planar projections, which are convenient enough for understanding, analysis and processing.

Practical formulations of mathematical problems of parallel computing, aimed at relevance and ease of application, mainly use the power of available IT to reproduce mathematical models that were previously limited by the low efficiency of equipment, and now have the prospect of becoming a new information revolution.

The well-posed formulations of the majority of parallel programming problems have developed on the basis of sequential algorithms and include a revision of the capabilities of the tools used, associated with the measure of organization of a previously created imperative program. Some complications are associated with using uniprocessor configurations as the base model for parallel multithreaded programs. It is possible that it is more useful to consider dual-processor configurations as a limiting case. The appearance of parallel algorithms proper is not so common, although in the mid-1990s conferences of this direction were held.

Teaching parallel computing is included in the educational programs of many universities, which is sufficient for understanding their complexity and setting the objectives of their research. The problem is the transition from the theory of parallel computing to the practice of implementing high-performance programs that satisfy especially complex, difficult to verify criteria of reliability and safety. The difference in the substantive complexity of the labor functions of a researcher and a developer of massively used programs is focused on the requirements aimed at understanding the problems being solved, their usefulness. A number of difficulties arise from the development of the areas of application of the obtained solutions, the determination of the limits of applicability of the results and the achievement of consistency in the generalization of the created tools, which are most important for solving the problems of modern IT.

The transition to experiments on supercomputers has shown that it is system solutions that can make a significant contribution to the performance of parallel computing, and such a contribution can exceed theoretical predictions. This can be seen as a rationale for the need for a more fundamental approach to programming, especially to systems programming and its mathematical foundations. A characteristic feature of the systems approach as the leading programming method is the transition to classes of problems in meaningful analysis of problem statements. Class boundaries are set when choosing a problem-solving

process. This makes it possible to create high-performance programs with partial use of the functional programming paradigm at the level of new and scientific problem statements [17–19].

When creating, forming and researching mathematical models as a fundamental basis for solving especially difficult problems of efficiency, reliability and safety of software, an important role is played by the development of models related to time and resources, which are poorly represented in classical mathematics courses. The qualification of a systems programmer includes the formation of the ability to independently invent solutions to new problems and the skills to responsibly improve the quality of ready-made solutions, along with a deep knowledge of non-classical and fundamental mathematics and familiarization with modern physics and the achievements of the humanities. This is too inconvenient for an educational system focused on persistent stereotypes and is contrary to methodological traditions.

3. PRINCIPLES OF FUNCTIONAL PROGRAMMING

Usually, functional programming implies the support of a number of semantic and pragmatic principles that contribute to the creation of functional models at the stage of computer experiments, useful in solving new problems. The programmer follows the semantic principles when preparing the program. The pragmatic principles are provided by the programming system, freeing the programmer from little significant solutions that do not depend on the nature of the problem. Most of these principles were laid down by J. McCarthy in the first implementations of the Lisp language [1].

3.1. Semantic principles

Functional programming supports semantic principles for representing algorithms, such as universality, parameter independence, self-applicability.

Universality. The concepts of “function” and “value” are represented by the same symbols as any data for computer processing. Each function applied to any given data, in a finite time produces a result or diagnostic message. A historically related concept is the stored program principle.

This principle allows you to build representations of functions from their symbol parts. and calculate the parts as data arrives. They can be formed even in the course of calculations and processing of information about them. There are no obstacles to handling function representations as well as handling data. In principle, there are no restrictions on the manipulation of language means, functions from the definition of the semantics of the language, constructions for the implementation of the language in the programming system, as well as expressions in the program. Everything that was needed when implementing a programming language can be useful when using it. This determines the openness of functional programming systems. Strictly speaking, unlike mathematics, programming doesn't deal with values or functions at all. Programming works with data that can represent values or functions. This was noted long ago by S.S. Lavrov [5, 9].

This is how compilers construct programs. When compiling programs, memory is allocated for functions, variables, and constants. The effectiveness of such a distribution depends on taking into account the peculiarities of the basic means of processing machine codes, which is usually formulated as a type of data that is convenient for computer processing, but somewhat contradicts the principle of universality.

Self-applicability. Function representations can use themselves directly or indirectly, which allows for the construction of clear concise recursive symbolic forms. The representation of both values and functions can be recursive.

Examples of self-applicability are given by many mathematical functions, especially recursive ones, such as factorial, Fibonacci numbers, series summation and many others, the definition of which uses mathematical induction. In programming technology, methods of step-by-step or continuous development of programs, as well as extreme programming, have some similarities. These methods reduce the organization of the programming process to a series of steps, each of which provides either a workable part of the program, or a tool for performing the next development steps.

Independence of parameters. The order and method of evaluating the parameters is irrelevant. The function parameters are independent of each other.

You can note that the parameters when calling the function are calculated at the same level of the hierarchy, in the same context. Some of the parameters are calculated before calling the function, others can be calculated later, but in the same context. Therefore, the representation of any highlighted formula from a function definition can be turned into a parameter of that function. This means that parts of the

function representation can be calculated depending on the intermediate results and functions can be constructed taking into account the conditions of their use, in particular, the location of their definitions and calls at different levels of the program representation hierarchy. Any symbolic form in a function definition can be extracted from it as a parameter and, conversely, substituted into it.

Data reuse is ensured by naming. Parameters have names, often called variables, although in functional programming they do not change values within the same context. Purely at the level of concepts, a variable is a named part of memory intended for multiple access to changing data, and a constant is to immutable data. Changing the relationship between a name and a value in functional programming is possible only by moving to another context, which is equivalent to changing the name. Functional variables are valid and equal to regular constant functions and can be argument values or generated as the results of other functions. The lack of skills in working with functional variables only means that it is necessary to master such an opportunity, the potential of which can exceed expectations now that programming is becoming more component-oriented.

3.2. Pragmatic principles

Functional programming supports pragmatic principles for computations, such as flexibility of constraints, immutability of data, and rigor of the result. Pragmatic principles are supported by the programming system, more precisely, the developers provide it.

Flexibility of constraints. On-line analysis of memory allocation and freeing is supported to prevent unreasonable memory downtime.

It happens that there is not enough memory, not for the entire task, but only for individual data blocks, perhaps of not so importance for its solution. Such a problem in functional programming systems is solved by the principle of flexibility of total constraints on spatial characteristics. Situations arise when some of these parts are exhausted, while others have underutilized space. In functional programming systems, this problem is solved by a special function – the "garbage collector", which tries to automate the reallocation or freeing of memory when any memory area is insufficient. This means that the given data can be of any size. New implementations of garbage collector efficiently take into account the advantages of bottom-up processes on large amounts of memory. Many new programming systems now include such mechanisms regardless of the paradigm.

Immutability of data. The representation of each result of function is placed in a new part of free memory without distorting the arguments of this function, which can be useful for other functions.

At any time, access to the data arising from the evaluations is possible. Thus, the debugging of programs is greatly simplified and the reversibility of any actions is ensured. You can be sure that all intermediate results are saved, they can be analyzed and reused at any time. If the definition of a function is a static construct, then the process can be viewed as a composition of functions, unfolded in dynamics according to this construct.

A separate aspect is associated with the transition from integers to real numbers, possibly requiring a change in the accuracy of the representation in the course of calculations. Logically, they remain constants, but the programming system treats them as variables.

Rigorous result. Any number of function results can be represented as a single symbolic form, from which the desired result can be selected if necessary.

Often this principle is interpreted as a requirement for a single or even exclusively scalar result of function, which leads to doubts about the legality of using integer division functions, root extraction, inverse trigonometric functions and many other categories of mathematical functions.

3.3. Consequences

The representation of algorithms in the form of functional programs gives practically significant consequences. Constructiveness, provability and factorization follow from semantic principles. Pragmatic principles lead to hidden models of the continuity of processes, reversibility of actions and unary functions. These consequences serve as the basis for the intuitive construction of functional models that make it possible to carry out and understand a direct computer experiment. In addition, a number of semantic and pragmatic principles support the preparation of functional models of programs for organizing parallel computations, which can be reduced to complexes of non-deterministic threads.

Meta-programming or constructiveness is a consequence of the principle of universality, which allows program representations to be processed in the same way as any data.

A given data representing a value or function can be built from parts down to letters. Any parts of the given can be computed fragments. Any function can act as a predicate for selecting a fragment to be substituted as part of a given one, as well as a parameter or function definition. This provides support for meta-compilation, including syntax-driven methods for generating and parsing programs. Also, homogeneous representations of programs are supported, externally preserving the analogy or similarity to the processed data or prototypes. Including mixed and partial computations are possible, optimizing transformations, macro-generation and much more, necessary for the creation of operating systems and programming systems.

The verification is based on the connection of the principle of self-determination with the methods of recursion, mathematical induction and logic.

It becomes possible to logically deduce individual properties of programs and, thanks to this, detect some subtle errors. If the representation of a given has a semblance of some inference logic, then its properties can be inferred using this logic. Most of the software verification systems are created within the framework of functional programming. This increases the reliability and security of programs, although it does not allow solving the correctness problem in full. Difficulties are associated with the insufficiency of classical logic in relation to non-classical logic of programming.

Factorization directly follows from the principle of parameter independence, taking into account the principle of universality.

Parts of the given, any subformulas are the equivalent of the function parameters. For any givens with one or a group of selected fragments, one can imagine a function, the parameters of which can be these selected fragments, after the substitution of which a given is obtained, which is equivalent to the original one. This allows the concept of selection or partial computation to be used. Thanks to factorization, it is possible to construct projections similar to the scheme admitting a proof.

Any marked set of program fragments can be removed from the given one, representing the program, and associated with a certain name, for allowing the possibility of restoring the original representation. You can note that the parameters when calling the function are calculated at the same level of the hierarchy, in the general context, according to the principle of data immutability. Therefore, the order in which the parameters are evaluated does not matter, it can be arbitrary. This is what makes it possible to decompose the program into being autonomously developed modules and accumulate correctness, as well as represent parallel threads, lazy or ahead of computations. We can say that the program can be presented in a factorized form according to various parameters, depending on the purpose of its transformation and further development. Due to the reversibility of actions, that is, the immutability of the data, the process of debugging the program acquires convergence, allows you to bring the program to the limit of compliance with the problem statement.

The consequences of supporting pragmatic principles in functional programming systems form intuitive images such as process continuity (infinity), reversibility of actions, and unary functions that provide the basis for building functional models. In addition, a complex of semantic and pragmatic principles provides support for the preparation of functional models of programs for organizing parallel computations, which can be reduced to complexes of non-deterministic flows.

Process continuity intuitively follows from the pragmatic support of the principle of flexibility of constraints.

After performing any function, you can execute another function. The STOP command is not a function. She has no arguments or results. It is just a signal to the processor to stop working. When performing any function, you can simultaneously perform other functions and before performing any function, other functions can be performed. This allows a significant part of the work to be performed on the basis of the unlimited memory model without much concern about its boundaries and the variety of characteristics of the speed of access to different data structures. In many languages of functional programming, imitation of work with infinite data structures is supported.

Reversibility of actions is based on the illusion of data immutability, the mechanisms of which are hidden in the programming system.

After executing any function, you can return to the point of its call. Any function can be repeated with the same parameters, or otherwise it can be interpreted in a different way, or any other function can be performed instead. Their application requires almost no care during program preparation and the bulk of debugging. Data changes that are actually needed, such as memory reuse, are simply automated. This allows you to support the mechanism for memorizing functions on previously processed arguments. The

programmer can afford not to tamper with the implementation of such facilities as long as there are no performance problems.

Unary function is based on the principle of a rigorous result and similarly allows a function of any number of arguments to be converted to a unary function with single argument.

For any function with an arbitrary number of parameters, you can construct its equivalent with one parameter. Since results are often arguments to enclosing functions, the accompanying principle of unary functions logically arises. In addition, the ability to go from a list of parameters or results to a single argument or strict result and back again allows you to move away from the usual scheme of operations, which maps two operands into the one result, to operations that map one set of operands into another set of results.

3.4. Applications to parallel computing

Parallelism arises from a mutual complex of semantic and pragmatic principles, which allows, if necessary, to consider any amount of presented data and, if necessary, reorganize the streams space. They make functional programming convenient for working with programs aimed at organizing parallel processes. First is the principle of parameter independence, which guarantees the same context when evaluating the parameters of a function of the same level. It becomes possible to represent independent threads and combine them into multithreaded or multiprocessor programs, into a common problem-oriented complex. In addition, parallelism uses the principles of strict result and universality, which allows considering any number of presented streams if necessary and reorganizing the stream space if necessary. The principles of pure functional programming are not quite convenient for modeling interacting and imperatively synchronized processes. A certain clarification of the principles arises, which simplifies the preparation of parallel computing programs.

Repression of a small probabilities is aimed at preventing an excessive number of threads corresponding to too unlikely probable situations, which somewhat narrows the principle of universality.

The principle of universality has two aspects - equal rights of programs and data and completeness of function definitions. When solving problems of parallel computing, universality preserves the aspect of equality of programs and data, traditionally in demand in the tasks of operating systems. The completeness of functions, convenient for building programs from debugged modules, can create problems due to the increase in the number of threads in multiprocessor programs associated with a variety of diagnostic situations. Any fragment, the calculation of which is unlikely or impossible, can be removed from the function representation, converted into a deferred action. Branches for practically irrelevant situations can be deleted or moved to the debug version. Sometimes this problem is overcome by choosing expressions that do not require branching, more often by checking data types. The amount of necessary diagnostics can be partially reduced by means of static analysis of data types.

Load balancing reduces the real execution time of the program, which can be viewed as a transfer of the principle of flexibility to time constraints.

Lazy or premature computations provide the ability to quickly redistribute the load. A complex function definition can sometimes be reduced to two functions, the first of which performs part of the definition, postponing the execution of the rest, and the second resumes the execution of the deferred part. Perhaps the execution of the second function will occur simultaneously with the second or even earlier.

Self-determination in the form of recursive functions is usually viewed as a complication that leads to dangerous stack growth. It should be noted here that many functional programming systems offer a number of practical solutions. These include deferred actions, memoization, ascending recursion, dynamic programming techniques, and optimization of recursions by reducing to loops, which in many cases make it possible to practically eliminate excessive stack swelling. And support for working with a stack within the framework of the principle of flexibility of constraints can be supported more efficiently than in most programming languages and systems. In addition, factorization of programs into scheme and fragments allows separating components according to the level of debugging complexity and inheriting the correctness of previously debugged modules. Functions are used that do not require preliminary evaluation of parameters, like macro technology. Similarly, in the mpC language, the amount of computation is redistributed when an uneven load of processors is detected [27].

Iterating space is used as the main parameter to control the parallelization of the loops.

Any finite set can act as an iteration space for a function defined on it. If there is a set of data on which the calculation of a function on one element does not require its results for other data, then using it as an

iteration space is convenient for the simultaneous execution of this function on all elements of this set. A similar technique for the distribution of operations and functions is available in the APL, Alpha and Sisal languages. The role of parameter independence is growing, it provides solutions to the problems of thread reorganization when tuning to different configurations of multiprocessor systems, requiring the decomposition of program fragments. The technique and concept of iterating spaces is convincingly supported in the Sisal language, in which iteration spaces are constructed over enumerable sets using scalar and Cartesian product operations [28, 29].

It is somewhat more complicated with pragmatic principles requiring a revision of system solutions at the level of programming systems development.

Automatic parallelization consists in extracting autonomous parts from the program that allow independent execution. If there is a function F with a known execution time T, which can be decomposed into two functions F1 and F2, such that the execution time of each of them is noticeably less than T. then if they are independent, then they can be executed in an arbitrary order, and the execution time of the original function will be less than T.

Identity of repeated runs for the purpose of debugging programs and measuring their performance.

The transition to supercomputers showed that with too many processors, the ability to observe program re-execution, necessary for debugging and measurements, disappears. During the next run, there may be failures on different processors. Functional programming here can allow special interpretations of the program, taking into account the protocols and the results of previously executed runs with tracking the execution identity.

Multi-pin fragments such as the control schemes or operations that have a number of operands and producing more than one result, at first glance, contradict the principle of a rigorous result.

However, the possibility of transition from a strict result allows one to build multithreaded functions that take parameters from a number of peer-to-peer threads and generate a number of results in terms of the number of threads. Thus, it is possible, as in the functional parallel programming language Sisal, to switch to operations that map one row of operands to another row of results [28]. This may correspond to the structure of some hardware nodes and thus allow more efficient solutions to be presented.

3.5. Productivity increase

In the transition to reusable programs and parallel computing, the *success of the application and the performance of programs become more important than their formal correctness and efficiency*. Pure functional programming can be viewed as a functional modeling technique for prototyping complex problem solving programs. The broader paradigm of production functional programming allows you to move from such functional models and prototypes to more efficient data structures, making practical decisions and trade-offs in their processing depending on real conditions when necessary. In addition to principles and consequences in real programming languages and systems, the production paradigm of functional programming allows for trade-off balancing mechanisms that look like special functions in the programming language. For example, Lisp 1.5, Clisp, Cmucl, and other members of the Lisp family typically provide these trade-off functions:

- *data type control* softens the principle of universality by functions of static and dynamic analysis of data types;
- *loops schemes* that simulate a slightly expanded variety of familiar control mechanisms for computations overcome typical concerns about the implementation complexity of the principle of self-determination;
- *data recoverity* when using destructive functions that have safe analogs makes it possible to eliminate excessive memory consumption, partly counteracting the principle of data immutability;
- *programmable predictions o*f memory volume and execution speed allow to programming memory allocation to neutralize sloppy flexibility limitations;
- *pseudo-functions,* in addition to generating the result, carry out a load in form the impact on external memory or interaction with devices, including I / O and file operations, that allows the principle of parameter independences;
- *memorization* allows you to radically reduce the complexity of repeated calculations by storing the results for all parameter values, which expands the principle of the rigor of the result. The

results of all streams have equal rights, all of them can be saved and reused without unnecessary calculations.

The general interaction of principles, consequences, applications and trade-offs in functional programming systems is presented in (see Table 2).

Table 2
Key aspects of the relationship between semantic and pragmatic mechanisms

	Semantics	Pragmatics
Principles	universality self-definition parameter independence	flexibility of constraints data immutability rigor of the result
Consequences	meta-programming verification factorization	continuity of processes reversibility of actions unary functions
Applications to parallel computing	repression of a small probabilities load balancing iteration spaces	automatic parallelization identity of repeated runs multi-pin fragments
Practical trade off	data type control loops schemes recoverity of data	programmable accounting for predictions pseudo-functions memoization

Within the framework of functional programming, it is possible to take into account the peculiarities of parallel computations that affect the choice of methods for their solution, depending on the priorities in the choice of language means and implementation possibilities.

Paradigmatic errors found during the operation of familiar programming systems on modern multiprocessor and network equipment show that some of them were simply invisible before the advent of networks, mobile devices and supercomputers. Consequences of the semantic and pragmatic principles of functional programming and the high modeling power of the apparatus of functions, extended with special functions of practical compromises, make it possible to usefully supplement the main paradigms of parallel computing and practice work to improve program performance.

4. Conclusion

In February 2021, the 22nd conference was held on modern trends in functional programming [30]. The presented reports convincingly showed the focus of functional programming on solving many problems of organizing parallel computations.

Some questions have not yet received a practical answer. The emergence of new paradigms can be associated with the manifestation of a range of new problems, the solution of which is still causing difficulties. It is not clear how appropriate the interaction of paradigms is. The educational problems of mastering new paradigms are left aside. In addition, the features of paradigms are only partially expressed at the level of program presentation. Others are requirements for the pragmatics of the system implementation support of the programming language.

Among them, the problem is associated with the poverty of language solutions for representing the discipline of access to multilevel heterogeneous memory and protocols of interaction between processes, which requires some refinement of the mechanisms of data immutability. Data immutability is maintained at the thread-local level, but causes problems when migrating to shared and external memory. Language tools for describing the discipline of access to multilevel heterogeneous memory and protocols of interaction between processes would be useful, which requires some refinement of the mechanisms of data

immutability. There are data blocks of different sizes, different access times to them, different service disciplines, possibly simultaneously available for different functions and storing copies of data. If data blocks are available for different functions, they can act as protocols, messages, and other representations of dependencies between functions. Usually such dependencies are represented in shared memory. It is possible that shared memory mechanisms require not so much data immutability as the ability to restore them, in addition to the mathematical aspects of working with heterogeneous memory, copies, replicas, etc., which is similar to dynamic editing of complex structures already illustrated in the tasks of working with DSL languages [30].

References

[1] J. McCarthy, LISP 1.5 Programming Manual / J. McCarthy. The MIT Press, Cambridge, 1963. 106 p.
[2] J. Backus, Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs. Commun. ACM 21, 8 (1978) 613–641.
[3] P. Khenderson, Funktsional'noye programmirovaniye. Moscow: Mir, 1983. 349 s.
[4] E. H'yuvenen, Y. Seppanen, Mir Lispa, t. 1, 2. Moscow: Nauka, 1994.
[5] Hudak P. Conseption, Evolution and Application of Functional Languages. ACM Computing Servys 21 3 (1989) 359–411.
[6] P. Graham, ANSI Common Lisp. Prentice Hall,1996. 432 p.
[7] S. S. Lavrov, Funktsional'noye programmirovaniye. Komp'yuternyye instrumenty v obrazovanii 2-4 (2002).
[8] S. S. Lavrov, L. V. Gorodnyaya, Funktsional'noye programmirovaniye. Interpretator yazyka Lisp. Komp'yuternyye instrumenty v obrazovanii 5 (2002).
[9] S. S. Lavrov, Metody zadaniya semantiki yazykov programmirovaniya. Programmirovaniye 6 (1978) 3–10.
[10] L. Gorodnyaya, On the presentation of the results of the analysis of programming languages and systems. CEUR Workshop Proceedings 2260 (2018) 152–166.
[11] L. Gorodnyaya, Method of paradigmatic analysis of programming languages and systems. CEUR Workshop Proceedings 2543 (2020) 149–158.
[12] F. P. Bruks, Kak proyektiruyutsya i sozdayutsya programmnyye kompleksy. Moscow: Mir, 1979. 152 s.
[13] P. Wegner. Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess. 1, 1 (August 1990), pp. 7–87. URL: https://pdfs.semanticscholar. http://dx.doi.org/10.1145/
[14] L. V. Gorodnyaya, Paradigma programmirovaniya: uchebnoye posobiye. Sankt-Peterburg: Lan', 2019. 232 s. URL: https://e.lanbook.com/book/118647 (data obrashcheniya: 12.11.2021)
[15] L. A. Zakharov, S. B. Pokrovskiy, G. G. Stepanov, S. V. Ten, Mnogoyazykovaya transliruyushchaya sistema. Novosibirsk, 1987. 151 s.
[16] Ch. Lange, Ontologies and languages for representing mathematical knowledge on the Semantic Web, Semantic Web 4 (2013) 119–158. https://doi.org/10.3233/SW-2012-0059
[17] L. Gorodnyaya, On parallel programming paradigms. CEUR Workshop Proceedings 1482 (2015) 587–593.
[18] V. A. Val'kovskiy, V. Ye. Kotov, A. G. Marchuk, N. N. Mirenkov, Elementy parallel'nogo programmirovaniya. Moscow: Radio i svyaz', 1983. 240 s.
[19] C. A. R. Hoare. Communicating sequential processes. Prentice-Hall International, UK, 1985.
[20] L. S. Baraz, E. V. Borovikov, N. G. Glagoleva et al., The Rapira programming language (Preprint). Siberian div. of the USSR Acad. of Sciences, Inst. of Informatics Systems, No. 4). Novosibirsk: Inst. of Informatics Systems, 1990.
[21] Stiven R. Palmer, Dzhon M. Felsing, Prakticheskoye rukovodstvo po funktsional'no-oriyentirovannoy razrabotke PO. Moscow: Vil'yams, 2002. 299 s.
[22] A. L. Fuksman, Tekhnologicheskiye aspekty sozdaniya programmnykh system. Moscow: Statistika, 1979. 184 s.
[23] G. M. Weinberg, The Psychology of Computer Programming. New York: Van Norstand Reinhold Comp., 1971. 279 p.
[24] V. L. Averbuh, Vizualizaciya programmnogo obespecheniya. Ekaterinburg: IMM UrO RAN, 1995. 168 s.
[25] A. P. Ershov, Kontseptsiya ispol'zovaniya sredstv vychislitel'noy tekhniki v sfere obrazovaniya (informatizatsiya obrazovaniya). Novosibirsk, 1990. 58 s. (Prepr. /AN SSSR. Sib. otd-niye. VTS; № 888).

[26] L. Gorodnyaya, Strategic Paradigms of Programming, Which Was Initiated and Supported by Academician Andrey Petrovich Ershov. Selected Papers 2020 5th International Conference on the History of Computers and Informatics in the Soviet Union, Russian Federation and in the Countries of Mutual Economic Assistance Council, SoRuCom 2020, 2020, 1–11, 946–972.
[27] A. L. Lastovetsky, mpC: A Multi-Paradigm Programming Language for Massively Parallel Computers. ACM SIGPLAN Notices 31 (2) (1996) 13–20.
[28] D. C. Cann, SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab., Livermore. California, May 1992. 128 p.
[29] V. N. Kasyanov, Sisal 3.2: functional language for scientific parallel programming. Enterprise Information. Systems 7 (2) (2013) 227–236.
[30] P. Koopman, S. Michels, R. Plasmeijer. Dynamic Editors for Well-Typed Expressions. Trends in Functional programming/ 22nd International Symposium, TFP 2021, February 17–19, 2021. Springer, LNCS 12834 (2021) 44–66.