Functional programming learning path

Lidia V. Gorodnyaya
A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences
Novosibirsk, Russia
Novosibirsk State University
Novosibirsk, Russia
gorod@iis.nsk.su

Dmitry A. Kondratyev
A.P. Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences
Novosibirsk, Russia
apple-66@mail.ru

The article describes one scheme of teaching functional programming, which has developed in many years of teaching experience on the basis of Mechanics and Mathematics Faculty of Novosibirsk State University. The issues of mastering functional programming are considered as a methodology for solving new and research problems of applied and system programming. It turned out to be useful for use the results of analysis and comparison of programming paradigms as distinguishing features of functional programming. Specifically, these are the priorities of decision-making at different stages of teaching programming and debugging programs, including the analysis of problem statements and options for their solutions. The current trend in the use of functional programming for the organization of parallel computing and functional modeling in solving applied programming problems is taken into account.

In general, the learning scheme is based on laboratory practice, familiarization with a number of functional programming languages and the principles formulation of the functional programming paradigm that distinguish it from other paradigms. As a result, the learning process becomes clearer, which gives structure to the system of learning tasks. Consequences are derived from these principles, showing the methods of successful functional programming application in solving complex problems, such as organizing parallel computing and improving the performance of programs created within the framework of functional programming paradigm. Attention is paid to the complexity of creating programs for solving new problems on the example of parallel computing. The requirements for an educational parallel computing language that supports the metaparadigm of functional programming are described. For educational and new research problems, the correctness of solutions is more important than the effectiveness of the programs obtained. It is this choice of priorities that allows functional programming to be considered as a general technique for preparing functional models both in the educational process and in the production of software tools. This suggests that functional programming serves as a training studio for programmers.

1 Introduction

Traditionally, the A. P. Ershov Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences, in cooperation with Novosibirsk State University, conducts research and development of methods for teaching programming. Part of this work is to create experimental courses for undergraduate and graduate students. Since the beginning of 1990s, a continuously developing course ”Functional Programming” has been delivered on the basis of Mechanics and Mathematics Faculty of

Submitted to:
TFPIE 2022

Novosibirsk State University. Since the late 1990s, courses ”Programming Paradigms” have been developed for the Faculty of Mechanics and Mathematics, the Higher College of Informatics and the Information Technologies Faculty of Novosibirsk State University. These courses contain the sections on functional programming as a meta-paradigm that supports the modeling of different paradigms and programming languages. The distance course ”Functional Programming Fundamentals” was created for the Internet University of Information Technologies [1] in 2004. Distance learning course ”Introduction to Lisp Programming” [2] and course “Programming paradigms” [3] were created in 2006. A new distance course ”Programming Paradigms” based on the Moodle system was created in 2019. The experience of teaching functional programming accumulated on the basis of these courses made it possible to form and test in practice a fairly reliable training scheme. This scheme is based on laboratory practice, familiarization with a number of functional programming languages, and the principles formulation of the functional programming paradigm that distinguish it from other paradigms. The formulation of principles made the learning process clearer and provided guidelines for the selection of learning tasks for laboratory work. Consequences were derived from clearly formulated principles that help to successfully apply functional programming in solving complex problems, using the problem examples of organizing parallel computing studied in other courses. The requirements for a universal educational multi-paradigm parallel computing language that supports the meta-paradigm of functional programming are described. The important idea that the correctness of programmed decisions is more important than the effectiveness of resulting programs is sharply emphasized . It is this choice of priorities that allows functional programming to play the role of a general methodology for teaching programming, including the preparation of functional models both in the educational process and in the production of software tools. This suggests that functional programming serves as a training studio for programmers.

The article contains a description of the main features of the functional programming paradigm and the functional programming training scheme, obtained on the basis of many years of experience on the basis of NSU. After the introduction in the second section of the article, a scheme for teaching functional programming to undergraduate and graduate students is given. The learning scheme is based on the description of the principles of functional programming presented in the third section, also provides examples of experiments with multithreaded programs In conclusion, a brief description of the course scheme itself is given and some unresolved issues are noted.

2 Skills, understanding, outlook

The general relationships between of principles, consequences, applications, and practical trade-offs in functional programming is based on the acquisition of skills in the application of semantic principles in the programming and debugging of programs and on an understanding of the pragmatic principles supported at the programming system. In addition, with the rapidly expanding space of programming languages and the development of a multitude of programming paradigms, it is necessary to familiarize students with the trends in functional programming, its prospects, its place among other paradigms.

First of all, during training, something similar to finger exercises is honed, which allows students to easily apply semantic principles in the programming process, such as universality, self-application, equal rights of parameters.This will allow new programmer to define debug-friendly generic functions that always produce a result, achieve laconic definitions through the use of recursion, and choose independent parameters with a clear representation of the boundary between bound and free variables.

Further, there are learning tasks that give an understanding of the prospects for use of
meta-programming, verification, the autonomy of the modules being developed as a consequence of

reducing the complexity of programming at the same time as increasing the reliability of obtained programs. Meta-programming allows young programmer to actively use and edit debugged templates, which is often more convenient than the direct application of formal rules. Verification shows the role of proofs and axiomatization in ensuring the correctness of programs with respect to the models and programming systems used. The autonomy of developed modules shows the possibilities of multidimensional combinatorics, accumulating the correctness of independently developed components and reusing them in different programs.

After that, a practical acquaintance with lazy evaluation, the identity of re-execution, iteration spaces as functional programming methods that affect the performance of calculations, the convenience of debugging programs, and the presentation of multithreaded programes is appropriate. Lazy evaluation allow programmer to optimally choose the moments for performing calculations. Identity of re-execution is an important debugging mechanism that allows students to compare the results of different runs of the programe. Identity of re-execution is an important debugging mechanism that allows students to compare the results of different program runs.

Further improvement in programming techniques is usually associated with an increase in the efficiency of programs - the speed of execution and the amount of memory required. To do this, control of data type is built into the programs, which allows programmer to specify the actual scope of the function definition. Recursive functions in case of performance problems be reduced to more efficient cycle diagrams. Memoization allows programmer to drastically reduce the time of calculations by storing the results of functions for the prevention of using them double executions, which is especially useful for complex data processing.

Pragmatics principles are supported by the programming system, which frees the programmer from solving problems that are not fundamental for the nature of tasks. However, in addition to familiarizing young programmer with the methods of their implementation, training works on modeling such methods, including inventing own solutions, are useful. In this regard, the mechanisms of restrictions flexibility can be reproduced on the processing of vectors. Data immutability can be shown on the mechanisms of wikipedia and continuous software development systems. The rigores of result shows its advantages when familiarized with the definition of abstract machines.

Pragmatic consequences are predominantly manifested in the preparation and debugging of programs, due to the assumption of process continuity, reversibility of actions, unary functions. Process continuation is easy to show in the description of processes whose boundaries are difficult to predict. Reversibility of actions is very helpful in debugging programs dynamically, using UNDO backtracking in editing systems. Unary functions are convenient for turning program fragments into uniform flows of actions.

Applications in different areas usually require specific solutions at the programming system level. For example, automatic parallelization, load balancing, multi-pin fragments can be useful for solving problems of organizing parallel computing. Not all programming systems have solutions for such problems, most often automatic parallelization is supported. Support for load balancing and multi-pin fragments is not common.

Facilities of communication with the outside world significantly affect the ability to debug and implement programs. Usually these are programmable forecasts, device access pseudo-functions, data recovery. Programmable forecast allows student to think about the actual boundaries of program execution, take into account the dependence on available resources, including time. Tools access pseudo-functions provide access to external devices, including I/O. Data recovery is usually organized as an correct alternative to data immutability, aimed in case of performance problems to transition debugged programs with immutable data to dosed datum changes without violating the correctness of their processing.

The acquired skills in applying the semantic principles of functional programming and understanding the methods of supporting pragmatic principles in functional programming systems are the basis for getting acquainted with a rather diverse range of solutions inherent in different functional programming languages. Of their many, special attention should be paid to historically significant and most popular languages. The idea of set available mechanisms obtained in this way allows us to determine the place of functional programming among other paradigms, to clearly understand its features. Here, exercises on the representation of programs for solving learning problems within different paradigms are useful, which is especially use to perform in multi-paradigm languages that support functional programming along with other paradigms.

3 FUNCTIONAL PROGRAMMING PRINCIPLES

3.1 Short history

Functional programming is one of the first paradigms aimed not so much at obtaining an efficient implementation of pre-created and well-studied algorithms, but at a complete solution of new and research problems [4, 5, 6]. In training, an important role is played by the convenience of preparing functional models of training programs, which make it possible to quickly debug the main mechanisms of program behavior in a simplified form. The turn of efficiency comes after the skills of achieving the correctness of solutions to the problem and assessing the feasibility of multiple application of programmable solutions have been obtained. In the pioneer era of the computers use, the practice of programming calculations according to known, previously created algorithms was characteristic, the correctness of which was not in doubt or was proven long before the appearance of first computers. Therefore, it was sufficient to ensure the program effectiveness. To substantiate the features of teaching functional programming, this article uses the results of the analysis of programming paradigms presented in [8, 7]. They stating that it is necessary to single out distinctive features that allow verification. It is this approach that serves as the basis for a stable learning process. As testable features, it turned out to be useful to use priorities in decision-making at different stages of the analysis of the problem statement, and then the development and debugging of the program for solving it. Accounting for priorities allows students to streamline the process of studying and searching for programmable solutions. For functional programming, priority is given to the organization of calculations and data structures, and issues of memory processing and process control are pushed to the periphery of attention.

The growth in the popularity of functional programming has so far occurred with a change in the element base, which gives rise to vast classes of new problems. The novelty of these tasks is due not only to the untapped capabilities of technical innovations, but also to the change of specialists generations, as well as the expansion of the circle of software users [9]. Functional programming helps to improve the skills of specialists due to the convenience of preparing programmable solutions to educational problems or their functional models. After debugging the functional models, it's time for a more efficient version of the program, perhaps created within a different paradigm. The skills of preparing and debugging functional models will be useful when working within any programming language. Of particular importance is the perspective of functional programming as a universal method for solving problems burdened by hard-to-verify and poorly compatible requirements. This allows, when teaching functional programming, to acquaint students with the methods of verification and specification of programs. Separately, the requirements for an educational language of parallel programming, which includes a sublanguage of functional programming, are described.

The term ”functional programming” currently allows two interpretations. Historically, in the early

1960s, J. McCarthy proclaimed that all programming concepts can be interpreted as functions or the result of function application [4]. In the mid-1970s, J. Backus drew attention to functional programming, calling for overcoming the narrowness of so-called “bottleneck” of the assignment operator with the help of functional programming languages [5]. The actual definition of the term was not given, it was used intuitively without any particular discrepancies and gradually began to be perceived as a separate programming paradigm, in which it was realized that correctness was more important than efficiency. Functional programming gives priority to the organization of calculations and the hierarchy of data structures, and the issues of allocation and processing of memory, as well as the management of calculation processes, go by the wayside. Such prioritization in the educational process is based on mathematical and linguistic models, using the skills of abstracting problem statements and programs.

Around the mid-1990s, the idea of ”pure functional programming” as a branch of discrete mathematics crystallized. The refusal to actively use various side effects associated with assignments, transfers of control and external devices, which complicate theoretical studies, was proclaimed. There was a reduction of programmable solutions to constant calculations. Pure functional programming can be seen as the mathematical basis for more general functional programming. This allocation roughly corresponds to the difference in standards for academic and industrial programming languages.

It can be noted that usually optimizing compilers replace constant calculations with a constant calculated during compilation, and duplicated formulas with a variable whose value will be calculated when the program is executed. Back in the early 1960s, J. McCarthy noted that the role of global variables can be played by the outermost local variables [4]. Modern purely functional programming languages, such as Haskell, extend the language with special ”monads” [10] to work with side effects and external devices, while others are included simply library modules or packages. The absence of control transfers in many functional languages is compensated by the introduction the concept of ”continuation”, which allows passing the name of next function through parameters [11]. Thus, in the transition to purely functional programming, there is not an “exclusion” of certain programming concepts, but a “change” in their form of entry into programs or a push to the periphery of attention.

Typically, functional programming implies the support sets of semantic and pragmatic principles that contribute to the creation of functional models that are useful in solving new and research problems.In preparing a program, a programmer may follow semantic principles that are partly expressed or proclaimed in the definition of a programming language. Pragmatic principles are provided by the programming system, freeing the programmer from decisions that are not related to the nature of the problem statement. Descriptions of such principles usually appear as notes when defining a language. Most of the semantic and pragmatic principles were laid down by J. McCarthy in the first implementations of the Lisp language [4]. Some developed later in the practice of developing new functional programming systems in the search for more efficient and productive system solutions [11]. When teaching programming, the role of principles is shown on not too difficult learning tasks and in comparison with other programming systems.

3.2 Semantic principles

Functional programming supports the semantic principles of function representation, such as universality, self-applying, equal rights of parameters.

Universatility. The concepts of ”function” and ”value” are represented by the same symbols as any data for computer processing. Each function applied to any data produces a result or a diagnostic message in a finite time.

This principle allows programmers to build representations of functions from their parts - symbols, and even calculate parts of the functions representation as data arrives. Learning tasks for the principle of universality can be selected from linguistic practice and symbolic data processing. In such tasks, both data and programs look like texts, they can be analyzed and processed like ordinary texts.

A historically close concept is the stored program principle. The idea of a stored programe was first formulated in the description of analytical engine by Charles Babbage, a hundred years later it was implemented in computers by Konrad Zuse and the definition of a machine by Alan Turing, and later proclaimed in the architecture of John von Neumann. As a result, the sufficiency of a universal representation of information was initially confirmed, in which there is not much difference in the nature of data for representing values and functions. Therefore, there are no barriers to processing data that represents functions in the same way that data that represents values is processed.

Self-applicability. Function representations can use themselves directly or indirectly, which allows the program designer to arrive at clear and concise recursive symbolic forms.

The representation of both values and functions can be recursive. Examples of self-applying give many mathematical functions, especially recursive ones such as factorial, Fibonacci numbers, summation of series, and many others whose definition uses mathematical induction. Too straightforward use of recursion often leads to excessive computation. Here it is appropriate to show different methods of implementing recursion. So it is possible to consider the reduction of recursion to cycles, lazy evaluation, auxiliary functions, memoization, dynamic programming and other methods.

Equal rights of parameters. The order and method of calculating the function parameters do not matter.

Function parameters do not depend on each other. It is not the order in which the parameters are calculated that matters, but only their correspondence to the names of parameters. Students can notice that the parameters when calling the function are calculated at the same level of the hierarchy, in the same context. Therefore, some parameters can be calculated both before calling the function and during its execution, most importantly, in the same context. Therefore, the representation of any distinguished formula from the definition of a function can be turned into a parameter of this function. This provides an additional ground for training exercises that will be useful later for the practice of program conversion. Here it is useful to show the technique of working with additional parameters, which often make it possible to increase the efficiency of calculations. Typical tasks can be seen in the ability to define functions for processing and interpreting lists, depending on the changing context. A historically significant example was given in his time by S. Kleene when solving the problem of reducing the function ”-1” to the function ”+1”, at the same time showing the benefit of bottom-up methodology in solving new problems.

Function variables are valid and equal to ordinary constant functions and can be the values of arguments or be formed as the results of other functions. The rarity of skills in working with functional variables means a challenge to the programming learning system. The potential of such skills may exceed expectations now that programming is becoming more component-oriented.

3.3 Pragmatic principles

Functional programming supports the pragmatic principles of organizing computations, such as flexibility of constraints, immutability of data, and strictness of the result. Pragmatic principles are supported by the programming system, more precisely, by its developers. Methods for supporting such principles are usually not defined in the programming language, they are chosen by the developer of programming system, usually based on tradition. It is convenient to introduce pragmatic principles in the form of describing ways to support them and showing the problems that arise in practice using many programming systems that do not support such principles.

Flexibility of restrictions. To prevent unreasonably busy memory, dynamic data reachability analysis, freeing and reusing memory that stores unreachable data is supported.

This principle provides a special function - ”garbage collector” (garbage collector), which tries to automate the reallocation or release of memory when some area of memory is not enough [4]. This means that the data can be of any size. New implementations of the garbage collector effectively take advantage of bottom-up processes on large amounts of memory [11]. Many modern programming systems now include such mechanisms regardless of the programming paradigm and language.

Data immutability. The representation of each function result is placed in a new piece of free memory without mangling that function’s arguments, they can be useful to other functions.

At any time, access to data previously obtained as a result of evaluations is possible. At any time, access to data previously obtained as a result of evaluations is possible. This greatly simplifies the debugging of programs and ensures the reversibility of any actions. There is always confidence that all intermediate results are saved, they can be analyzed and reused at any time. It is useful to introduce the technique of representing programs in the style of single assignment in any programming language. A similar program transformation is usually performed by optimizing compilers.

The rigores of result. Any number of function results can be represented in a single symbolic or structural form, from which the desired result can be selected or reorganized as needed.

Often this principle is interpreted as requiring a single or even exclusively scalar function result. This leads to doubts about the legality of using integer division, rooting, inverse trigonometric functions, and many other categories of mathematical functions. In this regard, G.M. Fikhtengol’ts, gave remarkable considerations in the preface to a textbook on mathematical analysis. He notes that if direct functions are often single-valued, then inverse ones, as a rule, do not have this property [12].

This principle is supported in many programming languages, the main difference lies in the restrictions on result structure. Functional programming does not restrict the data structure of the result.

3.4 Consequences

Representation of algorithms in the form of functional programs gives practically important consequences. Meta-programming, verification and autonomy of developed components follow from semantic principles. Pragmatic principles lead to intuitive models of processes continuity, reversibility of actions, and unary functions. These consequences make it possible to set up and understand a direct computer experiment and the development of programmable solutions, which inevitably arises when debugging even not very complex and especially new programs.

Meta-programming is a consequence of the universality principle, which allows programmers to process and create representations of programs in the same way as any data.

Data representing a value or function can be made up of parts upto letters. Any parts of a given can be computed fragments. In the old days, S. Pipert completed the description of Logo language with the call “Create your own programming language”. This inspiring task allows students to overcome the barrier to self-development of game, editing, interpreting and compiling programs. t is only important to dose the complexity so that the solution of problem fits into the regulations of educational process. Mixed and partial calculations, optimizing transformations, macro generation and more are possible. This can be done in almost any programming language, but there is usually no such tradition or clear examples.

Verification is based on the connection of self-applying principle with the methods of recursion, mathematical induction and logic.

Most program verification systems are created within the framework of functional programming. It becomes possible to logically deduce individual properties of programs and, thanks to this, to detect some subtle errors. If the representation of a given object is similar to some inference logic, then its properties can be inferred using this logic or similar models. The educational practice of using verification systems can show the dependence of correctness verification on axiomatics and encourage understanding of the role of the mathematical apparatus in programming.

The autonomy of developed modules directly follows from the principle of equal rights of parameters, taking into account the universality principle.

A striking example of the benefits of autonomy is provided by the history of separate modules compilation, which may have essentially determined the long life of the Fortran language. It should be noted that Lisp systems typically compile functions rather than complete programs. The frequency of modules use, procedures or functions in the development of new programs is usually much higher than the use of ready-made program. The increased frequency of use of constituent modules, like libraries of standard procedures, in practice causes a high level of trust, higher than verification.

Continuity of processes intuitively follows from the pragmatic support of the constraints flexibility principle.

After executing any function, every processor can execute any other function. (The STOP command is not a function. It has no arguments or results. It's just a signal to the processor to stop working.) While any function is being executed, other functions can be performed at the same time. This allows a significant part of the program work to be based on the model of unlimited memory without much concern about its boundaries and the variety of characteristics of the access speed to various data structures. Many functional programming languages support imitation of working with infinite data structures, the technique of which can be found on educational tasks similar to generating an infinite series of numbers [10].

The 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, the processor can return to the point of its call. Any function can be repeated with the same parameters, it can be interpreted differently or replaced by any other function. Applying action reversibility requires little to no care during program preparation and the bulk of debugging. This allows programe systems to support a function memoization mechanism that preserves the results for previously processed arguments. In fact, necessary data modifications, such as memory reuse, are simply automated. The programmer can afford not to interfere with the implementation of such facilities as long as there are no performance issues. It is enough to get acquainted with the methods and measure the effectiveness.

Unary functions are related to the principle of rigores result principle.

For any function with an arbitrary number of parameters, it is possible to construct its equivalent with one parameter, which is a structure of the original parameters. Since results are often arguments to enclosing functions, the mechanism of unary function that accompanies the strict result principle follows logically. This style of presentation of programs is useful when moving to multi-threaded program.

3.5 Parallel Computing Applications

Within the framework of functional programming, parallelism relies on a common set of semantic and pragmatic principles that allows one to consider independent streams of represented data and, if necessary, reorganize the stream space. Such a set of semantic and pragmatic principles makes functional programming convenient for working with programs aimed at organizing parallel computing, which can be reduced to complexes of independent, non-deterministic flows. First of all, this is the parameter independence principle, which guarantees the same context when calculating function parameters, regardless of the position in the argument list. It becomes possible to represent independent threads and combine them into multi-threaded or multi-processor programs, into a common software package. In addition, the principles of strict result and universality are used in parallelism. Pure functional programming is not very convenient for modeling interacting and imperatively synchronized processes. Therefore, in the future, some refinement of the principles will be required, which simplifies the preparation of high-performance parallel computing programs.

Lazy evaluation allows programm to push back unlikely calculations to avoid the cost of executing an excessive number of threads corresponding to too rare situations.

Any fragment that is unlikely or impossible to execute can be moved from a function view to a delayed action. Branches for almost irrelevant situations can be 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 volume of necessary diagnostics can be partially reduced by static analysis of data types.

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

The transition to supercomputers showed that with too many processors, the ability to reliably observe the repeated execution of a program, which is necessary for debugging and measurements, disappears. During the next run, there may be failures on other processors. Functional programming here can allow special interpretations of the program, taking into account the protocols and results of previously performed runs with tracking the identity of execution, as well as analysis of the programs execution with distortions. It is also possible to organize a comparative execution of the same program in two different threads on different processors. And support for working with the stack within framework of principle of constraints flexibility can be supported more efficiently than in most languages and programming systems [11]. Moreover, the factorization of programs into schemes and fragments allows us to separate components according to the level of debugging complexity and inherit the correctness of previously debugged modules. Functions that do not require preliminary calculation of parameters are used, for example, macrotechnics.

The iteration space is used as the main parameter to control the parallelization of loops in the absence of relationships between iterations.

Any finite set can work as an iteration space for a function defined on it. If there is a data set on which the evaluation of a function on one element does not require its results on other elements, then using such a set as an iteration space is convenient for simultaneously executing this function on all elements of the set. A similar technique for propagating operations and functions is available in APL, Alpha, BARS, and Sisal [13, 14]. The role of equal rights of parameters is growing, it provides a solution to the problems of threads reorganization when tuning to various configurations of multiprocessor systems that require program decomposition into schemes and fragments. The technique and concept of iteration spaces is strongly supported in the Sisal language, in which such spaces are built over enumerable sets using the operations of scalar and cartesian product [14].

It is somewhat more difficult with pragmatic principles that require a revision of system decisions at the level of development into an optimizing compiler. Here automatic parallelization plays an important role, load balancing and presentation of multi-pin fragments are rare.

Automatic parallelization consists in extracting autonomous parts from the program that allow independent execution.

If there is a function with known execution time that can be decomposed into two or more functions such that the execution time of each is appreciably less, then if they are independent, they can be executed. randomly or simultaneously, and the execution time of the original function may become shorter.

Typically, parallelization is performed by an optimizing compiler based on the results of static analysis.

Load balancing equalizes the real execution time of threads on different processors of a multiprocessor complex, which can be considered as an extension of the principle of flexibility to time constraints.

Lazy or lookahead calculations make it possible to quickly and efficiently redistribute the load. A complex function definition can sometimes be reduced to two functions, one of which executes part of the definition, deferring the rest, and the other resumes execution of the delayed part. Perhaps the execution of second function will occur simultaneously with the first or even begin earlier. In the mpC language, it is proposed to quickly redistribute the amount of computations when an uneven loading of processors is detected [15]. Load balancing is performed at the execution stage of the program based on dynamic analysis, possibly using instructions in the program to assess the uniformity of processor load and transform the executable code, which is no longer a rarity in new programming languages.

Multi-pins fragments, such as control circuits or operations that have multiple operands and give more than one result, at first glance, contradict the principle of strict result.

However, the ability to expand the understanding of the strict result principle allows programmers to build multi-threaded 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 set of operands to another set of results [14]. This may correspond to the structure of some particularly efficient hardware units and, thus, allow the presentation of more efficient solutions, taking into account the advances in development of element base. It is known that one of the arguments for RISK-architectures was the lag in efficiency of some particularly complex machine instructions from system equivalents of simple instructions. Now the ratio of efficiency has shifted sharply in favor of hardware solutions.

3.6 Productivity increase

With the transition to reusable programs and parallel computing, the performance of program becomes more important than their formal correctness. There is a change of priorities. Pure functional programming can be thought of as a functional modeling technique for prototyping complex problem solving. The broader paradigm of production functional programming allows programming process to move from such functional models to more efficient data structures, making practical decisions and trade-offs in their processing depending on real conditions. In addition to the principles and implications in actual programming languages and systems, production functional programming typically includes practical trade-off mechanisms that look like special functions in the programming language. For example, Lisp 1.5, Clisp, Cmucl. Clojure and other members of the Lisp family typically provide the following functions of this kind:

Data type control softens the principle of universality with the functions of static and dynamic analysis of data types.

Loop diagrams that model a somewhat extended set of familiar computational control mechanisms overcome typical concerns about the complexity of implementing the self-applying principle.

Memoization makes it possible to radically reduce the complexity of repeatedly repeated calculations, making an understandable concession to successful experience against the principles of rigorous result and verification.

Programming predictions of memory size and execution speed allow programmable memory allocation and load balancing, neutralizing the coarse mechanisms of the constraint flexibility principle.

Pseudo-functions , in addition to generating a result, perform actions on external memory or interact with devices, including input-output operations and file operations, which somewhat affects the principles of parameter independence and data immutability.

Recoverability of data when using destructive functions that have safe counterparts makes it possible to eliminate excessive memory consumption, partly deviating from the principle of data immutability.

Thus, within the framework of functional programming, educational practice on the organization of parallel computing can show the dependence of methods choice for programmable solutions on the priorities in language tools choice and implementation possibilities. Consequences of the semantic and pragmatic principles of functional programming and the high modeling power of the functions, complemented by special function of practical trade-offs, make it possible to complement the main paradigms of parallel computing for practical work on improving program performance.

3.7 Experiments demonstrating functional parallel programming principles

Let us consider the following task: the sum of cubes of non-negative elements whose indices are even. Let us note that the *map*, *reduce*, and *select* functions allows simplifying solutions of many similar problems and allow parallel execution of such solutions. A function-parameter of *map* and *select* should depend on element index to solve the considering problem. But function-parameter of *map* and *select* depends only on sequence element. Consequently, an important challenge is the generalization of the *map* and *select* functions.

The *reduce* function allows you to calculate the sum of elements. This function can apply the sum to the reducible sequence. But reducible sequence should be generated by a construction that generalizes the *map* and *select* functions. Students consider this task and may suggest different solutions.

The solution to the problem is the iteration space from the Sisal programming language. Loops generate a reducible sequence in the Sisal language. The sequence of even indices allows us to solve the considering task.

Students can execute Sisal programs using cloud parallel programming system (CPPS) [16, 17].

An iteration space can be created using Sisal triplets or the Cartesian product of triplets. A triplet defines an arithmetic progression.

The following construct is a triple construction expression: lower_bound..upper_bound..step [18, 19]

Examples of triplets are the following constructs:

- 0..n..1 is a sequence of natural numbers up to *n*;
- 0..7..-5 is an empty triplet;
- 0..n..2 is a sequence of even numbers up to *n*;
- 0..n-1..2 is a sequence of indices of even elements of an array of length *n*.

The following construct defines a Sisal loop controlled by a range:

$$\begin{gathered} for\ var\ in\ triplet\ do \\ returns\ reduction\ expr \\ end\ for \end{gathered}$$

where

- *var* is a variable;

- *triplet* is a triplet;
- *expr* is a reducible expression (it may depend on *var*);
- *reduction* is a reduction (for example, *sum of*, *array of*).

The semantics of a loop controlled by a triplet may be described as follows:

- Each iteration corresponds to each triplet element (*var* corresponds to the value of the triplet element).
- The value of a single iteration is the value of the reducible expression.
- The value of a loop is the value of a reduction.
- A reduction is applied to the value of previous iterations and to the value of the current iteration.

The formal definition of this semantics is implemented in the C-lightVer system. Students can apply deductive verification approach to Sisal programs using the C-lighVer system [20, 21, 22, 23].

The solution of the considering problem is the following function:

```
function sum_elements_even_indices(
                     a: array of integers
                     n: integer returns integer)
  for i in 0..n-1..2 do
  returns sum of
            if (a[i] >= 0)
               a[i]*a[i]*a[i]
               else 0
            end if
  end for
end function
```

where

- `0..n-1..2` is a triple that defines a sequence of even index values of elements of an array of length `n`;
- `if (a[i] >= 0) then a[i]*a[i]*a[i] else 0 end if` is a reducible expression.

This function is written in the Sisal programming language. This task demonstrates comparation of *map-reduce-select* approach and Sisal loops to students.

We suggest students to use symbolic method of verification of definite iterations [24] for Sisal program deductive verification [20, 21, 22, 23]. Symbolic method of verification of definite iterations is applied to special kinds of loops (definite iterations). Body of a definite iteration is executed once for each element of data sequence. Symbolic method of verification of definite iterations allows defining inference rules for these loops without invariants. Symbolic replacement of definite iterations by recursive functions (function *rep*) is the base of this method.

We use ACL2 [25] as theorem prover in the C-lightVer system [20, 21, 22, 23]. Obtained verification conditions with applications of recursive functions correspond to ACL2 logic based on computable recursive functions. Automation of proving these verification conditions is based on using ACL2 system. The input language of ACL2 system is Applicative Common Lisp.

Let us consider the precondition of considering function written in the language of ACL2 system:
`(and (integerp n) (< 0 n) (equal (length a) n) (integer-listp a))`

Predicate *integerp* checks whether its argument is an integer. Predicate *integer-listp* checks whether its argument corresponds to list of integer elements.

The postcondition is `(= result (reduce-sum-even-indices n a))`. Special term *result* correspond to value of Sisal expression. We have defined *reduce-sum-even-indices* function using language of ACL2 system. Consequently, students define specifications using Applicative Common Lisp.

The loop expression in considering function is translated into recursive function *rep*. Replacement of *result* term in postcondition by application of *rep* results in the following verification condition:

```
(implies
  (and (integerp n) (< 0 n) (equal (length a) n) (integer-listp a))
  (equal
   (rep (triplet 0 (- n 1) 2)) a)
   (reduce-sum-even-indices n a)))
```

ACL2 successfully proved it by induction on *n*. Let us note that considering function is based on iterations and *reduce-sum-even-indices* is recursive function. This verification task demonstrates correspondence between iterations and recursion to students.

4 Concluding Remarks

The 22nd TFP conference dedicated to modern trends in functional programming was held [26] at February 2021. We believe that our scheme of teaching functional programming may give students not only the base but also understanding modern trends especially in the field of parallel computing. The presented reports convincingly showed the focus of functional programming on solving many problems of organizing parallel computing.

Our scheme for teaching functional programming is as follows:

Semantic decisions. Programming solutions to educational problems for the formation of skills in the application of semantic principles.

Pragmatic solutions. Familiarization with the methods of supporting pragmatic principles in functional programming systems.

Review of functional programming languages.

Experiments with the presentation of a multi-threaded program based on functional programming, as well as with the use of tools that allow improving the performance of programs and measuring the resulting performance.

Reproduction of programmed solutions based on another paradigm in a multi-paradigm programming language.

Some questions have not yet received a practical answer. Among them is the problem associated with the poverty of language solutions for representing the discipline of access to multilevel heterogeneous memory and protocols for interaction between processes, which requires some refinement of data immutability mechanisms. Datum immutability is supported at the thread-local memory level, but causes problems when moving to shared and external shared memory for different threads. It is usually convenient to represent dependencies between functions, and hence between threads, in shared memory.

In this regard, the precedent with creation of the languages Cmucl and Clojure is interesting, giving a rebuttal to the explanation of reasons for criticism of functional programming by inefficiency and lack of connection with production systems. Cmucl is quite competitive with the C++ language in terms of efficiency, but this somehow did not attract much attention to it, which confirms the independence of functional programming purposes from the efficiency of its implementation. Clojure is implemented over the JVM, giving access to all modern IT features. Strictly speaking, functional programming can play the role of not only a training studio for the growth of professional programmer corps qualifications, but also a design department for production programming.

This article presents a description of the main features of the functional programming paradigm and the functional programming training scheme obtained on the basis of many years experience. The article contains a description of the main features of teaching functional programming. In the introduction to this article, a brief summary of many years of experience in teaching functional programming on the basis of Novosibirsk State University is given. Further, in the second section, a scheme for teaching functional programming to undergraduate and graduate students is given, containing a description of the main types of educational work and additions to the actual study of functional programming techniques with an overview of functional programming languages and the relationship of functional programming with other paradigms. The scheme is based on the description of the principles of functional programming, presented in the third section. It gives a brief outline of the history of the emergence of functional programming, shows the differences between semantic and pragmatic principles, consequences from them, features of specification for parallel computing and methods for improving program performance, and also provides examples of experiments with multithreaded programs. In conclusion, a brief description of the course scheme itself is given and some unresolved issues are noted.

5 Acknowledgements

I thank all my students of Mechanics and Mathematics Faculty of Novosibirsk State University for inspiring me to find a better way to introduce functional programming. I especially thank the organizers and participants of the conferences "Scientific service in the Internet environment", regular discussions, which helped to formulate the principles of functional programming for educational purposes.

References

[1] Lidia Gorodnyaya (2004): *Functional Programming Fundamentals*. Available at `https://intuit.ru/studies/courses/29/29/info`.

[2] Nikita Berezin & Lidia Gorodnyaya (2006): *Introduction to Lisp Programming*. Available at `https://intuit.ru/studies/courses/1026/158/info`.

[3] Lidia Gorodnyaya (2006): *Programming paradigms*. Available at `https://intuit.ru/studies/courses/1109/204/info`.

[4] John McCarthy (1963): *LISP 1.5 Programming Manual*. The MIT Press, Cambridge, USA.

[5] John Backus (1978): *Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs*. *Commun. ACM* 21(8), p. 613-641, doi:10.1145/359576.359579.

[6] P. Henderson & D.A. Turner (1982): *Functional Programming and its Applications: An Advanced Course*, First edition. Cambridge University Press;, USA.

[7] Peter Wegner, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2001): *How to Design Programs: An Introduction to Programming and Computing*, First edition. MIT Press, Cambridge, MA, USA.

[8] Lidia Gorodnyaya (2020): *Method of Paradigmatic Analysis of Programming Languages and Systems*. In: *Proceedings of the 21st Conference on Scientific Services & Internet (SSI-2019)*, CEUR Workshop Proceedings, Sun SITE Central Europe, pp. 149-158, `http://ceur-ws.org/Vol-2543/rpaper14.pdf`.

[9] Lidia Gorodnyaya (2020): *Strategic Paradigms of Programming, Which Was Initiated and Supported by Academician Andrey Petrovich Ershov*. In: *2020 Fifth International Conference "History of Computing in the Russia, former Soviet Union and Council for Mutual Economic Assistance countries" (SORUCOM)*, IEEE, Moscow, Russia, pp. 1-12, doi:10.1109/SORUCOM51654.2020.9464972.

[10] Paul Hudak (2000): *The Haskell School of Expression: Learning Functional Programming through Multimedia*. New York: Cambridge University Press.

[11] A.J. Field & P. Harrison (1988): *Functional Programming (International Computer Science Series)* , First edition. Addison-Wesley, USA.

[12] G.M. Fikhtengolts (1965): *Fundamentals of Mathematical Analysis: v. 1* , 1st edition. Elsevier , USA.

[13] Vadim E. Kotov (1980): *On Basic Parallel Language*. IFIP Congress 1980, USA.

[14] D. C. Cann (1992): *SISAL 1.2: A Brief Introduction and tutorial)*, 1st edition. Lawrence Livermore National Lab. , Livermore California, USA.

[15] Alexey Lastovetsky . (1996): *mpC: a multi-paradigm programming language for massively parallel computers*. *ACM SIGPLAN Notices* 31(2), p. 13-20, doi:10.1145/226060.226064.

[16] Victor Kasyanov, Elena Kasyanova, Alexandr Malishev (2021): *Support tools for functional programming distance learning and teaching*. In: *International Conference "Marchuk Scientific Readings 2021" (MSR-2021)*, Journal of Physics: Conference Series, IOP Publishing, Article ID 012052, doi:10.1088/1742-6596/2099/1/012052.

[17] Victor Kasyanov & Elena Kasyanova (2019): *Methods and System for Cloud Parallel Programming*. In: *Proceedings of the 21st International Conference on Enterprise Information Systems*, INSTICC, SciTePress, pp. 623–629, doi:10.5220/0007750506230629.

[18] Victor Kasyanov, Alexandr Stasenko (2009): *Sisal 3.2 Language Structure Decomposition*. In: *Proceedings of the European Computing Conference*, LNEE, Springer, pp. 533–543, doi:10.1007/978-0-387-85437-3_53.

[19] Victor Kasyanov, (2013): *Sisal 3.2: functional language for scientific parallel programming*. *Enterprise Information Systems* 7(2), p. 227–236, doi:10.1080/17517575.2012.744854.

[20] Dmitry Kondratyev, Alexei Promsky (2019): *Proof Strategy for Automated Sisal Program Verification*. In: *Software Technology: Methods and Tools*, LNCS, Springer, pp. 113–120, doi:10.1007/978-3-030-29852-4_-9.

[21] Dmitry Kondratyev, Alexei Promsky (2019): *Correctness of Proof Strategy for the Sisal Program Verification*. In: *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, IEEE, Novosibirsk, Russia, pp. 641–646, doi:10.1109/SIBIRCON48586.2019.8958225.

[22] Victor Kasyanov, Elena Kasyanova, Dmitry Kondratyev (2020): *Formal verification of Cloud Sisal programs*. In: *Applied Physics, Simulation and Computing (APSAC 2020)*, Journal of Physics: Conference Series, IOP Publishing, Article ID 012020, doi:10.1088/1742-6596/1603/1/012020.

[23] Dmitry Kondratyev, Alexey Promsky (2020): *Towards verification of scientific and engineering programs. The CPPS project*. *Journal of Computational technologies* 25(5), p. 91-106, doi:10.25743/ICT.2020.25.5.008.

[24] Valery Nepomniaschy (2005): *Symbolic method of verification of definite iterations over altered data structures*. *Programming and Computer Software* 31(1), p. 1-9, doi:10.1007/s11086-005-0001-0.

[25] J. S. Moore (2019): *Milestones from the Pure Lisp Theorem Prover to ACL2*. *Formal Aspects of Computing* 31(6), p. 699-732, doi:10.1007/s00165-019-00490-3.

[26] Pieter Koopman, Steffen Michels, Rinus Plasmeijer (2021): *Dynamic Editors for Well-Typed Expressions*. In: *Trends in Functional Programming*, LNCS, Springer, pp. 44–66, doi:10.1007/978-3-030-83978-9_3.