DOI: 10.14529/jsfi230203

The Semantic Model Features of the Statically Typed Language of Functional-dataflow Parallel Programming*

Alexander I. Legalov[1], ***Nickolay K. Chuykin***[1]

The features of a statically typed functional-dataflow model of parallel computation and its mapping to the statically typed language of functional-dataflow parallel programming Smile are considered. To provide support for architecture-independent parallel programming, we used: a functional style, an implicit managing of calculations on data readiness, structured data objects that provide representation of various types of parallelism. A distinctive feature of the approach is the inclusion in the model of special asynchronous data objects that can generate events on partial filling. These data objects are stream and swarm. Each of these data objects has its own specifics to control by parallel calculations. A stream is used to process data of the same type that arrives sequentially and asynchronously at random intervals. A swarm is used to describe independent data of the same type or different types, on which it is possible to perform massive parallel operations. The use of streams and swarms in various situations as well as their mapping into each other and other program objects are shown. An analysis is made of the possibilities of transforming the formed language constructs into programming languages used in writing programs for modern parallel architectures.

Keywords: parallelism, parallel computation model, architecture-independent parallel programming, functional-dataflow parallel programming, transformation of parallel programs.

Introduction

The modern development of parallel programs is focused on the use of methods that take into account the specific architectures of target computing systems. This is due to the desire to improve the performance of parallel computing. Research in the field of architecture-independent parallel programming has not yet formed to the final practical solutions and is carried in the following areas:

- automatic or semi-automatic parallelization of sequential programs with their subsequent transformation to the target architecture [1];
- development of programs or algorithms that has unlimited parallelism, determined by the problem being solved, with the subsequent "compression" of this parallelism in accordance with the restrictions determined by the target architecture [2].

There is a semantic gap between the written program and the real parallel computing system (PCS) with using any of these approaches. There is a loss of efficiency and balance, when program is transformed into machine code, since the characteristics of the "architecture-independent" serial or parallel algorithm conflict with the organization of calculations in specific PVS. That is why unlimited parallelism is often manually "compressed" when fine-tuning the program to suit the features of the computer, that defines an approach opposite to parallelization of sequential programs. Manual transformation leads to a loss of efficiency in the process of developing parallel software and do not allow to write a program once and for different architectures. In this regard, the problem is actual of searching for models of parallel computing and building on their basis programming languages and tools that provide effective transformation of the parallelism of a once written program for various computing resources.

*The paper is recommended for publication by the Program Committee of the International Scientific Conference "Parallel computational technologies (PCT) 2023".

[1]Higher School of Economics, National Research University, Moscow, Russian Federation

One of the ways that determines a more efficient transformation of architecture-independent programs to programs for real architectures is the use of static data typing, which provides efficient compilation into a typeless representation at the level of the command system architecture [3]. This approach is widely used in both sequential and parallel programming. However for writing architecture-independent parallel programs only static typing is not enough. It is also necessary to take into account the features of the constructions that describe parallelism, because their dynamic characteristics can make it difficult to transform into machine code for existing parallel computers.

The article is organized as follows. Section 1 is devoted to the analysis of the factors that determine the architectural independence of parallel programs. Section 2 discusses the features of a statically typed functional-dataflow parallel computing model. In Section 3 we present the semantics of interpretation statements for various relationships between data and functions. The conclusion summarizes the results of the study and indicates directions for further work.

1. Determining Factors of the Architectural Independence of Parallel Programs

Support for architectural independence from real computing resources in the description of parallel processes is generally provided both by the peculiarities of the representation of data storage methods and by the use of appropriate strategies for controlling of calculations [4].

The independence of data storage from memory is supported by a functional paradigm focused on representing programs as interacting functions. In contrast to the imperative approach, the data memory is presented in an implicit form. Using recursions instead of iterations allows you to get rid of the reuse of variables at the level of describing algorithms. These solutions largely ensure the architectural independence of programs and are implemented in various functional programming languages (FPL) [5]. However, most of these languages have limitations for the implicit representation of parallelism. That is due to the peculiarities of data structure organization as lists with sequential access to their elements. The presence in the list of access only to the head and tail does not allow to organize parallel computing directly in many modern functional programming languges. Therefore, to present concurrency in programming languages, explicit control of computations is usually used, on the basis of which threads or processes are created [6]. This leads to directive impact on concurrency of the programmer and is a factor that makes it difficult to port programs to other parallel architectures.

Dataflow control strategies allow to describe parallelism implicitly. One of the first such computational models is the Dennis model [7]. It formed the basis of a number of specialized data flow processors with different architectures. There are languages that use dataflow control and are released for various architectures. For example: Sisal [8], Colamo [9], LuNA [10]. A number of these programming systems combine data flow management with a functional style. However, for many computational models it is problematic to talk about architectural independence, which is often associated with the orientation of programming languages and methods for their transformation to certain architectural solutions. Most of them have not fully developed the concept of unlimited parallelism. Also, dataflow control is often combined with the use of explicit management or with the need to manage limited resources.

Along with the functional approach and dataflow control, some problems linked with an architecture-independent representation of parallelism can be solved using special data struc-

tures that not only contain data, but also support their various parallel behavior. The differences in the behavior of these data structures determine the approaches to the different organization of parallelism. Encapsulation of the behavior of these data structures inside of specialized dynamically generated and synchronized objects allows you to remove from programs the explicit control of calculations, which is usually used in imperative programming languages, replacing it by interaction with functions and other data structures with dataflow control. The use of parallel recursion allows us to consider the program as a description of activities performed in unlimited computing resources, formed implicitly as needed. Such an approach at the programming language level allows using it as an architecture-independent language. At the same time, this imposes special requirements on the transformation of programs from such languages into architecture-dependent programs.

The use of special data structures was implemented in the Pifagor functional-dataflow parallel programming language, which is based on the functional-dataflow parallel computing model (FDPCM) [11, 12]. These structures are represented as data lists, parallel lists, delayed lists, asynchronous lists. Each type of list defines its own methods for grouping data and dataflow control. The disadvantages of the proposed constructions and the language, include the dynamic typing of atomic types, as well as the dynamic formation of lists in the calculations process. It does not allow an effective output representation to be formed during program compilation.

The need to use static typing was confirmed during the implementation of a number of projects on the transformation of functional-dataflow parallel programs into real architectures:

- when converting to FPGA topology [13];
- when transforming into a statically typed imperative programming language [14].

To obtain the required solutions, it was necessary to use additional descriptions into the intermediate representation generated by the Pifagor compiler that define the types of processed data, and to limit the semantics of lists representing parallel structures.

Thus, to solve the problem of efficient transformation of an architecture-independent parallel program into a program for a real target architecture, it is necessary to jointly use approaches that do not solve the problem separately:

1. dataflow control;
2. functional programming paradigm;
3. special data structures designed to represent different types of parallelism;
4. static data typing.

Their joint application will be reflected both in the parallel computing model and in the tools created on its basis. Orientation towards architectural independence leads to the formation of a domain-specific parallel computing model and a domain-specific architecture-independent parallel programming language with a specific set of data structures and their semantics.

2. Statically Typed Functionally-dataflow Parallel Computing Model

The application of the considered approaches makes it possible to form a statically typed functionally-dataflow parallel computing model (STFDPCM). Borrowing many of the ideas from the previously proposed FDPCM [11, 12], the STFDPCM is oriented towards the use of a static type system and fixed dimensions of data objects. This, in turn, leads to a change in the semantics of program-forming operators. The axioms of the model and its transformation algebra are also

changed. They get oriented to a more efficient transformation at compile time. At the same time, the main characteristics of the model that determine the specifics of functional-dataflow parallel programming remain unchanged:

- calculations take place inside unlimited resources, which allows you to implicitly describe parallelism without resource conflicts;
- calculations are managed using dataflow control;
- the choice of operations and axioms that define the basic set of functions is focused on a visual textual representation of the information graph of the program during its subsequent description using a programming language;
- the computational model defines the general structure of a functional-dataflow parallel program without reference to operational semantics, which can be additionally defined, thereby defining the specifics of a particular language of functional-dataflow parallel programming.

The model is given by the triple:

$$\mathbf{M} = (\mathbf{G}, \mathbf{P}, \mathbf{S_0}),$$

where **G** is an acyclic oriented graph that defines the information structure of the program (its informational graph), **P** is a set of rules that determine the dynamics of the model's operation (the mechanism for generating markup), $\mathbf{S_0}$ – initial labeling of informational graph arcs, on which data has already been generated that determines the execution dynamics.

Informational graph of the program:

$$\mathbf{G} = (\mathbf{V}, \mathbf{A}),$$

where **V** is a set of vertices that define operators, and **A** is a set of arcs that define ways of information transfer from data source operators to receiver operators. Each source operator can be associated with one or more receiver operators. The receiver operator can have multiple inputs, each of which can be associated with the output of only one arc. The information transmitted along the arcs can be of any acceptable type, determined by the characteristics of the data objects used in the STFDPCM.

The vertices of the graph corresponding to the operators provide information transformations of data and their structuring in various ways. There are the following types of vertices:

- interpretation operators (interpreters);
- copy operators (denotations);
- operators for grouping to data structures (data objects);
- delaying computations operator (delay).

The use in model of static typing instead of dynamic typing imposes its own limitations, but, on the other hand, it provides additional opportunities for transforming functionally streaming parallel programs into programs for real architectures.

2.1. Interpreters

Interpreters are designed to perform functional transformations. Each such operator has two inputs, one of which receives a value perceived as a function of **F** (functional input), and the other (data input) receives a value that is an argument of **X**. The focus of model on compile-time analysis has led to two varieties of interpreters: single-argument operator and group-argument operator.

A single-argument interpretation operator, denoted in textual representation, as in FD-PCM [11], by "`:`" (postfix form) or "`^`" (prefix form), is designed to define ordinary functions that take an argument as a whole. The group-argument interpretation operator is used to set calculations on each element of a group data object, generating at the output a similar data object with elements whose type corresponds to the type of the result of the function being executed. Denoted by the double character "`::`" for postfix or "`^^`" for prefix forms, respectively.

Using different interpreters allows to unambiguously apply a function with the same name depending on the context. For example, the subtraction function "`-`" on the argument `(10,-3)`, perceived it as a vector consisting of two integers, generates the following results:

- $(\mathbf{10}, -\mathbf{3}) : - \Rightarrow \mathbf{13}$ – subtraction function of second number from first number;
- $(\mathbf{10}, -\mathbf{3}) :: - \Rightarrow (-\mathbf{10}, \mathbf{3})$ – sign change group function.

The division of the interpretation operator into single-argument and group-argument allows you to introduce a flexible set of additional functions for lists of various structures, while providing more diverse processing of asynchronously incoming data.

2.2. Copy Operator

Each vertex of the graph **G** admits the presence of several output arcs, along which the same value is transmitted to other vertices. This set of arcs can also be thought of as a single **copy operator** that transfers data from its single input to multiple outputs. In general, a copy operator can be combined with the preceding operator from which its output arc emerges. A chain of copy statements is also possible, which can be thought of as a single statement. In textual form, it is determined by naming the value that determines the output of the vertex, and further using the introduced designation at the required points of the informational graph. Both postfix naming of the propagated object in the form: "`value >> name`" and its prefix equivalent, which looks like: "`name << value`" are used. For example:

```
y << F^x;
(x,y):+ >> c;
```

The type of the denotation is the same as the type of the result of the preceding computations and is determined at compile time.

2.3. Data Objects

Data objects are designed to represent various options for grouping data into structures that have certain properties and behavior. It is them that define various options for representing parallelism and ways to determine the dataflow control during calculations. Data objects include:

- a constant operator (constant) that ensures the use of immutable values;
- synchronous grouping operator or **join** operator, designed to collect all incoming data together before their subsequent issuance as a result;
- operator of asynchronous grouping or **swarm**, performing ordering by input number, but not synchronizing data coming to different inputs;
- an asynchronous sequence operator or **stream**, that arranges data according to the time it arrived and produces as result the first arrived value.

Each kind of objects has its own behavior in the course of the formation of incoming data, exposing the result obtained to its output, transmitted along the arcs to the vertexes that receive information.

2.3.1. Constant

Constant operator or **constant** defines a node that stores a constant value and is always ready to execute. This operator has no input. The output is initially set to markup that defines the prescribed value. The set of constant operators of the informational graph form the internal initial markup of the computation model. In textual representation, a constant operator is given by a value of the corresponding type. The type of the constant is assumed to be known at compile time. Constants include data of various basic types, for example: integers, booleans, signals, atomic functions, as well as functions that are formed when writing a program. Functions are referred to as constants because their descriptions are fixed and are constructions that are immediately ready for execution. Constant examples:

- **10** – integer constant;
- **true** – boolean constant;
- **!** – signal constant (denoted by symbol "!");
- **+** – atomic function plus;
- **min** – the function name which developed for finding the minimum of two elements.

2.3.2. Join

Join has multiple inputs and one output. It provides structuring, ordering and synchronization of data coming from various sources along input arcs. The types of incoming elements must be known at compile time. The order of elements is determined by the numbers of inputs, each of which corresponds to a natural number in the range from **0** to $\mathbf{N-1}$, where **N** is the length of the generated data set. A connector signals ready when it receives all input. In text form, the operator is specified by delimiting the list elements with parentheses "`(`" and "`)`". For example:

```
(x0, x1, x2, x3)
```

The numbering of elements starts from zero and is set implicitly in accordance with their order from left to right. The element type of the join must be known at compile time and determines one of the possible interpretations: a vector or a tuple.

A join as **vector** is intended for grouping elements of the same type. This allows you to access its elements by index, as well as perform bulk operations on all elements.

The join as **tuple** also has multiple inputs and one output. It provides structuring, ordering and synchronization of heterogeneous data coming along arcs from various sources. The types of incoming data must be known at compile time. Elements are accessed by an index specified by a constant, which allows the type of the output value to be determined at compile time. In text form, similarly to a vector, it is specified by limiting the elements of the list with parentheses "`(`" and "`)`". Tuples can be used as arguments to functions, each of which, in its description, uniquely maps the element type to its location in the tuple.

2.3.3. Stream

The concept of **stream** extends the idea of the previously proposed asynchronous list [12]. The basic idea associated with asynchronous data arrival is preserved. However, all elements are assumed to be of the same type, which in turn cannot be a stream or a swarm. This is quite consistent with the concepts of universal statically typed languages. A stream can be considered as an object (Fig. 1), the main characteristics of which are:

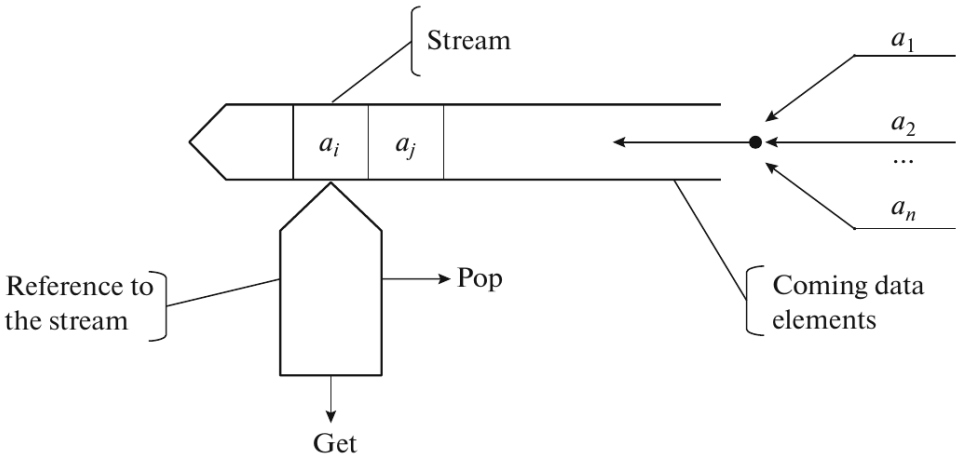

Figure 1. General scheme of a stream

- when at least one ready data element appears in the stream, it generates a signal informing about its readiness;
- the ready element can be read from the stream for processing;
- if, during processing of an element selected from the stream, new data items enter it, they can also be asynchronously selected from the stream in order of arrival and processed in parallel;
- parallel processed elements of the stream may arrive after processing in another stream, the type of which is determined by the type of the result of the function (in this case, the order of their arrival may differ from the initial one, depending on the processing time);
- we can check the stream for end of data in it and terminate the work if it is true.

In text form, grouping into a stream is specified by delimiting its elements with the symbols "`<[`" on the left and the closing square bracket "`]`" on the right. For example:

```
<[x0, x1, x2, x3]
```

The same type of stream elements is explained by the fact that if they appear randomly in time, it is impossible to determine the type of the current value at compile time.

2.3.4. Swarm

Swarm, unlike **join**, groups independent data. The arrival of each element in the swarm is accompanied by the issuance of a ready signal, informing the nodes of the informational graph about this event, receiving information from it. This allows you to quickly and asynchronously respond to changes in the state of the swarm.

In text form, grouping into a swarm is specified by limiting its elements with square brackets "`[`" and "`]`". For example:

```
[x0, x1, x2, x3]
```

Each element of the swarm is formed independently and is ready for execution when it appears. Like the connector, swarm elements can be of the same type, forming a vector, or of different types, forming a tuple.

A swarm, like a stream, allows you to process incoming data one element at a time (Fig. 2). This is possible due to the fact that the arrival of elements in the swarm can occur non-

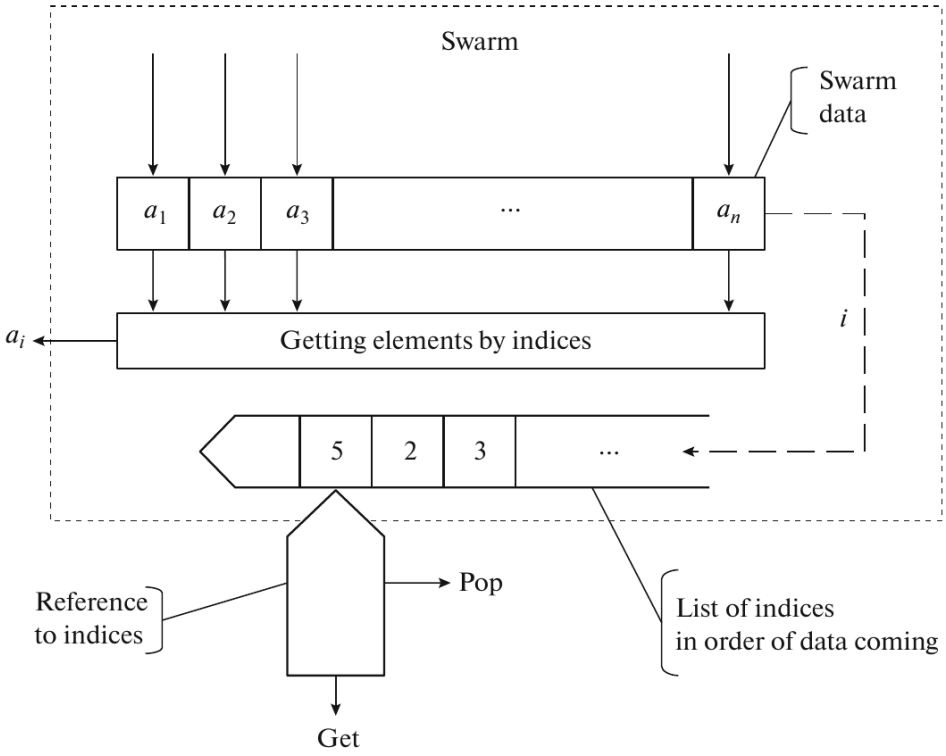

Figure 2. General scheme of a swarm and its reference

simultaneously and asynchronously. Therefore, the indexes of the elements can line up in the internal stream in the order in which they were received. Sequential selection of indices from this stream allows accessing swarm elements at the moment they arrive.

2.4. Delay

Delay operator or delay is specified by a vertex containing a valid informational subgraph that includes several input arcs and one output arc. The input arcs determine the arrival of arguments, and the output specifies the result issued from the subgraph. A specific feature of this grouping is that vertices bound by the delay operator cannot be executed, even if all arguments are present at their inputs. Their activation is possible only when the delay is removed (opening the contour), when the limited subgraph becomes part of the entire computed graph.

Initially, the delayed subgraph creates on its only output a constant markup, which is the image ("icon") of this subgraph. This markup propagates along the arcs of the graph from one operator to another, multiplying, entering various data objects and getting out of them until it arrives at one of the inputs of the interpretation operator. As soon as the delay operator becomes one of the arguments of the interpretation operator, instead of the "icon" the delayed subgraph is substituted with the input connections preserved. The contour of the delay operator encircling the subgraph is "removed" in this case, and the activated operators are executed. As a result, the resulting labeling is again formed on the output arc of the expanded subgraph, which is one of the arguments of the interpretation operator that expanded the delayed subgraph. This procedure is called delayed subgraph expansion.

In text form, the delay operator is specified by enclosing other operators with curly braces "{" and "}". For example:

```
{(a,b):+}
```

If it is necessary to form several independent arguments within the delay, then they are grouped into a swarm, which is initiated upon revelation:

```
{[x0, x1, x2, x3]}
```

The presence of a delay allows you to postpone the start of some calculations or not start them at all. It is necessary when organizing selective data processing. In addition, this operator, if necessary, can be used as brackets that change the priority of operator execution. To do this, it can be directly provided as one of the arguments to the interpretation operator.

3. Relationship between Grouping and Interpretation Operators

Calculations are formed when data arrives at interpretation operators. In this case, a calculation result is formed, the type of which depends both on the types of the argument and function, and on the type of the interpretation operation. Various combinations of these three components form the operational semantics of the computational model, and also determine options for equivalent transformations in accordance with the algebra of the model.

3.1. Using a Single-argument Interpretation Operator

A single-argument interpretation operator in most cases defines the traditional functional transformations of its arguments into a result. The following combinations of relationships between arguments are allowed:

- scalar – scalar;
- join – scalar;
- swarm – scalar;
- stream – scalar;
- scalar – join;
- join – join;
- swarm – join;
- stream – join;
- scalar – swarm;
- join – swarm;
- swarm – swarm;
- stream – swarm.

In these relations the first argument acts as the data to be processed, and the second defines the function that processes this data. Note that stream cannot act as a function. An argument is a scalar if it has a predefined base (atomic) type or is a constant of one of the base types.

3.1.1. Relation "scalar – scalar"

This relation is perceived by a single-argument interpretation operator as a traditional function of one argument. That is, the data being processed are constants or computed values of the underlying type. The generated result is determined by the semantics of the data interpreted

as a function. The function can be either predefined or developed by the programmer. For example:

$$\mathbf{x} : - \equiv -\mathbf{x}$$
$$\mathbf{x} : \mathbf{sin} \equiv \mathbf{sin}(\mathbf{x})$$

3.1.2. Relation "join – scalar"

The relation is almost the same as executing a function from several arguments, the values of which are defined as elements of the join:

$$(\mathbf{x}, \mathbf{y}) : + \equiv +(\mathbf{x}, \mathbf{y}) \equiv \mathbf{x} + \mathbf{y}$$
$$(\mathbf{x}, \mathbf{y}) : \mathbf{min} \equiv \mathbf{min}(\mathbf{x}, \mathbf{y})$$

3.1.3. Relation "swarm – scalar"

The relationship allows to form an asynchronous flow of arguments to the function that displays them. It is assumed that when any element appears in the swarm, the interpretation operator will launch a function that processes the swarm, which will perform a partial calculation. That is, in this case, there is no preliminary synchronization of arguments before calling the function. The situation is similar in many ways to using of the **inline-function**. It is assumed that such calculations are possible when input parameter of function is described as a swarm. Example:

$$[\mathbf{x}, \mathbf{y}] : \mathbf{min}$$

3.1.4. Relation "stream – scalar"

The relation determines the transfer to the input of a function the stream, the readiness of which is determined by the appearance of any first element. The analysis of data readiness for subsequent elements of the incoming stream is formed within the function that processes this stream.

It should also be noted that, in the general case, the number of incoming elements is not defined for streams. Therefore, along with the value generated in the stream, a ready flag is also generated, which is a boolean value. As a result, a join of the following format is formed at the output of the stream:

$$(\mathbf{value}, \mathbf{flag})$$

Therefore, before using the value at the beginning, it is necessary to check the truth of the flag, indicating that the required values from the stream are still coming. The last value that signals the end of data is the **false** flag. However, the data itself is no longer defined.

3.1.5. Relation "data – join"

The data can be: scalar, join, swarm, stream. The join as a functional argument specifies the simultaneous execution of all the functions over the processed argument. In this case, parallelism is implemented with many independent command streams over one data set. For any input data $\mathbf{X}$ and a list of functions $\mathbf{F} = (\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}})$ during interpretation, an equivalent transformation into a set of parallel executable statements is performed:

$$\mathbf{X} : (\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}) \equiv (\mathbf{X} : \mathbf{F_0}, \mathbf{X} : \mathbf{F_1}, ..., \mathbf{X} : \mathbf{F_{n-1}})$$

The output join is formed as result. For example, the simultaneous execution of addition, subtraction, multiplication, and division operations on a single data argument can be described as follows:

```
(x,y):(+, -, *, /) ⇒ ((x,y):+, (x,y):-, (x,y):*, (x,y):/)
```

3.1.6. Relation "data – swarm"

The options for using a swarm as a set of functions are similar in many ways to using a join in that role. The difference is that its elements can process the input argument without common synchronization. Therefore, the appearance of results of the interpretation operator that implements this relation can be arbitrary with the formation of a new swarm as result. The input data $\mathbf{X}$ and the list of functions $\mathbf{F} = [\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}]$ are transformed into a set of parallel operators:

$$\mathbf{X} : [\mathbf{F_0}, \mathbf{F_1}, ..., \mathbf{F_{n-1}}] \equiv [\mathbf{X} : \mathbf{F_0}, \mathbf{X} : \mathbf{F_1}, ..., \mathbf{X} : \mathbf{F_{n-1}}]$$

As result a swarm is formed. Performing addition, subtraction, multiplication, and division above the same data argument at parallel would look like this:

```
(x,y):[+, -, *, /] ⇒ [(x,y):+, (x,y):-, (x,y):*, (x,y):/]
```

3.2. Using the Group-argument Interpretation Operator

The group-argument interpretation operator is focused on the processing of group data objects by one function. Its main task is the simultaneous processing of elements located in data objects. In this case, the final calculations are formed through single-argument interpretation operators, reduction to which is carried out by applying equivalent transformations. The following combinations of relations are allowed for the interpretation group operator:

- join – scalar;
- swarm – scalar;
- stream – scalar.

More complex combinations of various data objects lead to the formation of multidimensional structures and are not yet considered at the level of the computation model.

3.2.1. Relation "join – scalar"

In this relation, the join is considered as a vector of elements of the same type, each of which is applied to the same function, acting as a scalar. Let $\mathbf{X} \equiv (\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}})$ be a join data object that defines the vector of processed data, $\mathbf{f}$ be a function. Then, taking into account the subsequent equivalent transformations, the group interpretation operator for this relation can be represented as follows:

$$\mathbf{X} :: \mathbf{f} \equiv (\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}) :: \mathbf{f} \equiv (\mathbf{x_0} : \mathbf{f}, \mathbf{x_1} : \mathbf{f}, ..., \mathbf{x_{n-1}} : \mathbf{f})$$

3.2.2. Relation "swarm – scalar"

This relationship is largely similar to the previous one. The difference lies in the fact that the execution of functions on individual elements begins immediately upon receipt of these elements. A swarm is also asynchronously formed as a result of the execution of functions.

The swarm is defined as follows: $\mathbf{X} \equiv [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}]$. A group operation on a swarm using the $\mathbf{f}$ function is defined by the following equivalent transformation:

$$\mathbf{X} :: \mathbf{f} \equiv [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}] :: \mathbf{f} \equiv [\mathbf{x_0} : \mathbf{f}, \mathbf{x_1} : \mathbf{f}, ..., \mathbf{x_{n-1}} : \mathbf{f}]$$

3.2.3. Relation "stream – scalar"

Group interpretation of the relationship is carried out by analogy with the previous ones. That is, the function is executed on each element of the stream as long as the stream receives elements from various sources. As a result, a new stream is formed at the output. It should be noted that the order of the results in the new stream may not match the order of the original data. This is because even when the same function is executed, the calculation time may differ for various reasons. The specificity of the stream is also that the number of processed elements may be unknown in advance, and the completion of the arrival of elements is determined automatically by reading the end-of-stream marker.

$$\mathbf{X} :: \mathbf{f} \equiv< [\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{n-1}}] :: \mathbf{f} \equiv < [\mathbf{x_0} : \mathbf{f}, \mathbf{x_1} : \mathbf{f}, ..., \mathbf{x_{n-1}} : \mathbf{f}]$$

Conclusion

The key idea of the approach is the initial attempt to abandon traditional architectural solutions and consider parallel computing as an architecture-independent domain specific area. This idea largely coincides with the views of J. Backus on whether programming can be liberated from von Neuman style [15]. The main difference from the previously presented works [2, 11–14] on the functional-dataflow model of parallel computing is the explicit separation of data objects from its program-forming operators and the adaptation of their semantics to a more complete analysis at the compilation stage. This should ensure efficient transformation into programs for various architectures of real parallel computing systems.

The presented base features of the statically typed model of functional-dataflow parallel computing formed the basis of the developed statically typed language of functionally-dataflow parallel programming Smile [16]. The proposed constructions make it possible to describe not only static data structures, but also to form various options for the dynamics of their behavior, which provides various additional possibilities for representing parallelism in programs. Options for representing dynamically changing parallelism appear, what depends on the relationship between the intensity of data arrival and the time of their processing. At the same time, along with data parallelism, their dynamic pipelining is possible through the use of streams [17]. Accounting for the use of static typing at the level of the computational model makes it possible to build a more efficient basis for subsequent transformations of architecture-independent parallel programs.

References

1. Steinberg, B.Ya., Steinberg, O.B.: Program transformations as the base for optimizing parallelizing compilers. Program Systems: Theory and Applications 12:1(48), 21–113 (2021).

https://doi.org/10.25209/2079-3316-2021-12-1-21-113 (in Russian)

2. Legalov, A.I., Vasilyev, V.S., Matkovskii, I.V., Ushakova, M.S.: A Toolkit for the Development of Data-Driven Functional Parallel Programmes. In: Sokolinsky, L., Zymbler, M. (eds) Parallel Computational Technologies. PCT 2018. Communications in Computer and Information Science, vol. 910, pp. 16–30. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99673-8_2

3. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)

4. Legalov, A.I.: On the control of computations in parallel systems and programming languages. Scientific Bulletin of NSTU 3(18), 63–72 (2004) (in Russian)

5. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)

6. Charpentier, M.: Functional and Concurrent Programming: Core Concepts and Features. Addison-Wesley (2022). 528 p.

7. Dennis, J.B., Fosseen, J.B., Linderman, J.P.: Data flow schemas. In: Ershov, A., Nepomniaschy, V. (eds) International Symposium on Theoretical Programming. Lecture Notes in Computer Science, vol. 5. Springer (1974). https://doi.org/10.1007/3-540-06720-5_15

8. Kasyanov, V.: Sisal 3.2: functional language for scientific parallel programming. Enterp. Inf. Syst. 7(2), 227–236 (2013). https://doi.org/10.1080/17517575.2012.744854

9. Levin, I., Dordopulo, A., Gudkov, V., *et al.*: Software Development Tools for FPGA-Based Reconfigurable Systems Programming. In: Voevodin, Vl., Sobolev, S. (eds) Supercomputing. RuSCDays 2019. Communications in Computer and Information Science, vol. 1129. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36592-9_51

10. Malyshkin, V., Perepelkin, V.: The PIC Implementation in LuNA System of Fragmented Programming. The Journal of Supercomputing 69(1), 89–97 (2014). https://doi.org/10.1007/s11227-014-1216-8

11. Legalov, A.I.: Functional language for creating architecturally independent parallel programs. Vychislit. Tekhnol. 10(1), 71–89 (2005) (in Russian)

12. Legalov, A.I., Redkin, A.V., Matkovsky, I.V.: Functional-dataflow parallel programming with asynchronously incoming data. In: Parallel computing technologies (PaVT'2009): Proceedings of the International Scientific Conference, Nizhny Novgorod, March 30 – April 3, 2009, pp. 573–578. Chelyabinsk, Ed. SUSU (2009) (in Russian)

13. Romanova, D.S., Nepomnyashchiy, O.V., Ryzhenko, I.N., *et al.*: Parallelism reduction method in the high-level VLSI synthesis implementation. Trudy ISP RAN/Proc. ISP RAS 34(1), 59–72 (2022). https://doi.org/10.15514/ISPRAS-2022-34(1)-5

14. Vasilev, V.S., Legalov, A.I., Zykov, S.V.: Transformation of Functional Dataflow Parallel Programs into Imperative Programs / Automatic Control and Computer Sciences 56(7), 815–827 (2021). https://doi.org/10.3103/S0146411622070239

15. Backus, J.: Can programming be liberated from von Neuman style? A functional stile and its algebra of programs. CACM 21(8), 613–641 (1978). https://doi.org/10.1145/359576.359579

16. Legalov, A.I., Legalov, I.A., Matkovskii, I.V.: Specifics of Semantics of a Statically Typed Language of Functional and Dataflow Parallel Programming - Scientific Services & Internet 2019. In: CEUR Workshop Proceedings, vol. 2543, pp. 274–284. https://doi.org/10.20948/abrau-2019-08

17. Legalov, A.I., Matkovskii. I.V., Ushakova, M.S., Romanova, D.S.: Dynamically Changing Parallelism with Asynchronous Sequential Data Flows. Automatic Control and Computer Sciences 55(7), 636–646 (2021). https://doi.org/10.3103/S0146411621070105