# Creation of Efficient and Portable Parallel Programs

**6. AUTHOR(S)**  
Prof. Larry Wittie

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**  
The Research Foundation of  
State University of New York  
Computer Science, SUNY Stony Brook  
Stony Brook, NY 11794-5000

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**  
Office of Naval Research  
Math, Computer & Information Sciences Div.  
Code 311/Dr. Andre M. van Tilborg, Director  
800 North Quincy St.  
Arlington, VA 22217-5660

**11. SUPPLEMENTARY NOTES**  
Approved for public release. Distribution is unlimited.

**13. ABSTRACT (Maximum 200 words)**  
This grant supported research to develop methods for parallel programming by masses of people using computers on networks. Scientific results have surpassed initial expectations. The convergence of little-known Russian developments in program transformation, supercompilation, and partial evaluation with the truly American phenomenon of Java programming have produced surprisingly strong results. This part of the pNet (programming parallel Networks) project is the pJava, or parallelizable Java, language. Particularly exciting is the development of program transformation techniques for machine understanding of parallel programs written in the familiar syntax of the important new Web programming language Java. The remainder of the research centers around the Norma language, developed to help applied mathematicians create efficient fluid-flow codes. Norma helps produce parallel programs that can run efficiently on large classes of parallel and distributed computers, including the Web. The short eight months of research supported by this grant have produced many sound scientific results. This work points the way to a grand unification of techniques in program transformation, parallelization, and compilation that will allow the creation of libraries of reusable parallel codes that will run efficiently on almost any computer system to be found in or on the computer networks of the world.

**14. SUBJECT TERMS**  
Parallel Computer Languages,  
Portable Parallel Programs for the Web, Efficient Reusable Code, Java,  
Program Transformations, Supercompilation, Norma, Fluid-Flow Codes

**17. SECURITY CLASSIFICATION OF REPORT**  
UNCLASSIFIED

**18. SECURITY CLASSIFICATION OF THIS PAGE**  
UNCLASSIFIED

**19. SECURITY CLASSIFICATION OF ABSTRACT**  
UNCLASSIFIED
The Principal Investigator made one trip to Moscow Russia in August 1996 and spent many hours on the internet in connection with this grant to develop methods for parallel programming by masses of people using computers and groups of computers on the network. Two Russian scientists visited New York for several weeks of collaboration and familiarization with American research facilities. These trips strengthened the scientific collaboration that had developed during eight prior trips to Russia by the Investigator to find Russian computer experts with technical results that could further world science if they were allowed to continue their research amid the gales of economic reform in Russia.

Scientific results have surpassed initial expectations. The Norma language, first developed to help applied mathematicians create efficient fluid-flow codes offers hope for producing parallel programs that can run efficiently on large classes of parallel and distributed computers, including the World Wide Web. Even more exciting is the development of program transformation techniques for machine understanding of parallel programs written in the familiar syntax of the important new programming language Java. Supercompilation techniques for very-high-level improvements of programs can work for Java programs, allowing masses of users to combine library routines in simple ways and yet produce highly efficient parallel codes.

The primary goals for the pNet project were: (1) To explore new principles, languages and methods for creating efficient portable parallel programs for massively parallel computers and networks of computers; and (2) To make it easy for technical application experts to specify elegant solutions to well understood, but possibly computationally complex information processing problems. The work on the Norma language has addressed these goals.

The secondary goals were: (1) To allow the creation and control of massively parallel(p) programs, run on possibly millions of cheaply available computers within huge geographically distributed networks(Net); (2) To simplify the creation of complex parallel programs using compositions of pretested program modules; (3) To make recent Russian developments for parallel computations available to scientists in western nations; and (4) To build lasting collegial and collaborative ties between American and Russian scientists.

The ONR sponsored series of personal scientific interactions between American scientists (including this Investigator) and computer scientists in the institutes of the Russian Academy of Sciences in Moscow and Novosibirsk have led to an unanticipated convergence of American and Russian computing cultures. Combining little-known Russian developments in the program transformation disciplines of supercompilation and partial evaluation with the truly American phenomenon of Java programming have produced surprisingly strong scientific results. The Russians taught the investigator about program improvements based on computer understanding and simplification of human produced programs. The Investigator suggested that the methods might speed up routinely interpreted Java programs without sacrificing the Java guarantees of safety
from rogue programs. In 1995, most Russian scientists had not then heard of Java. The result is a major part of the pNet project: that of developing the pJava, or parallelizable Java, language and program creation tools based on it. To produce these results, the Investigator and his Russian colleagues have had to share the chaos and privations of the Russian economic collapse, bridge long-instilled cold-war chasms of fear and distrust, and overcome an awkward language barrier. The results of the collaboration at both the scientific and human levels have been worth the discomforts.

Much of the first six months of funding for this project were spent climbing the mountains of paperwork required to support five Russian researchers for eight months in 1996. The two-week trip to Moscow in August allowed the investigator to help build modern personal computer systems for research on Norma, pJava, and supercompilation at two research centers in Russia, Keldysh Institute of Applied Mathematics in Moscow and the Program Systems Institute in Pereslavl-Zalessky. Both institutes are part of the Russian Academy of Sciences. More importantly, it allowed his Russian colleagues to keep their hopes alive after waiting in vain four months to be paid for their work on the project. The trip also permitted the investigator to acquire identical dual English-Cyrillic keyboards for use in both Russia and the United States by team members. During their trip to the New York in November, his two visitors installed operating system software compatible with that on their machines in Russia. This seemingly small detail of having the same keyboard and compatible personal computer software has made possible the efficient production of joint scientific reports.

The bulk of the scientific developments from this project are contained in the six technical reports attached to this final report. Together they cover nearly two hundred pages of analyses in applied mathematics and computer science. Three of the reports are newly written for this project. Three are translations of Norma design documents previously available only in Russian. The rest of this report gives the abstracts of the six technical papers in the Appendix.

Program Transformations for Java

For the investigator, the most exciting work is given in the paper entitled, "Program Transformations for Java" by Andrei V. Klimov of the Keldysh Institute for Applied Mathematics in Moscow, Valentin F. Turchin of The City College of New York, and Larry D. Wittie of SUNY at Stony Brook in New York. It shows how supercompilation can be used to transform parallel and distributed programs written in (p)Java and make them much more efficient. It also gives clear examples of how a Java supercompiler works to understand the deep meaning of (p)Java programs and can produce simpler equivalent programs.

A program transformation system for Java is presented. Two program understanding and transformation disciplines are reviewed: supercompilation and partial evaluation. Supercompilation is more general and powerful: partial evaluation is a subset, but simpler to understand and to use.

Transformation methods were originally developed for functional languages. Here, for the first time, is a demonstration of supercompilation for the imperative language Java. We show how
the main phases of a supercompiler work for Java: configuration, driving, configuration analysis. How to supercompile Java is explained via a Producer-Consumer example, including a trace of manual supercompilation steps.

The strong points of this report are:
- It gives an excellent review of program transformation methods;
- It shows that the syntax of the popular Java programming language can be used to write computer-understandable portable parallel programs that automatically can be deeply analyzed, optimized and tailored to run efficiently for many specific problems or computer system architectures; and
- Its major scientific contribution is a demonstration that supercompilation techniques are already powerful enough to fuse several concurrent threads into one, dramatically increasing the efficiency of some computer programs.

Supercompilers can perform significant inter-thread analysis and source code optimization. We explain why and how we plan to develop and implement a prototype supercompiler for Java.

Effectively a supercompiler can deduce the essential meaning of calls to many code functions and incorporate that meaning into highly-restructured faster-running codes.

pJava - A Parallel Superset of Java

The second report covers the planned steps in the development of an implicitly parallel version of the Java programming language, called pJava. The paper is entitled, "pJava - A Parallel Superset of Java for Automatic Parallelization" and was written by Andrei V. Klimov of the Keldysh Institute for Applied Mathematics in Moscow and Larry D. Wittie of SUNY at Stony Brook in New York.

An extension of Java, called pJava, which allows for automatic parallelization and efficient program transformation, is presented. The idea of pJava is to select a Java subset with suitable properties, and then gradually to extend it while preserving the properties. The starting point for pJava is a purely functional subset of Java with data limited to immutable objects. Higher-level declarative notions will gradually be added to allow for easy, reliable, implicitly parallel programming. Ultimately mutable objects will be allowed as well, but under a special programming discipline. This discipline can be satisfied at a low-level by a qualified Java programmer, or at a high-level via constructs based on monotone objects, which can be easily applied by less-experienced users.

The pJava language will allow programs to be written that can be ported to new computer systems and new application areas, and still be analyzed and improved automatically by supercompilation system algorithms. More rapid execution can result both from program transformations to produce faster, simpler, guaranteed-equivalent source codes and from automatic parallelization to use many computers at once on parts of the same code.
Creation of Efficient and Portable Parallel Programs

The third report is entitled, "Creation of Efficient and Portable Parallel Programs" and was written under the direction of Igor B. Zadykhailo of the Keldysh Institute for Applied Mathematics in Moscow. It gives a detailed semantic definition for the Norma parallel programming language for applied mathematics. This precise definition is needed to build the program-analysis systems that underlie both supercompilation transformations and automatic parallelization of mathematical solution algorithms for problems in applied mathematics.

The first publications appeared in late 50s - early 60s. The main idea of the language later named NORMA is very simple. It was an attempt to automate the design of the programs based on the jobs prepared by applied mathematicians from Keldysh Institute of Applied Mathematics for further programming. Usually those jobs were the result of applying numerical methods (more often grid method) to physical problems’ solution. The intention was to create the language of jobs’ specification corresponding to the constructions obtained after mathematical solution of the problem.

We have gained much experience in designing complicated program systems and translators, now it is time to create really friendly programming languages. We shall take a decisive step and turn from universal languages to the languages for users to formulate problem solution in generic terms. We are sure that universal languages may be friendly only to system programmer. Hence we shall bend every effort to creation of specialized language for each application domain.

As the specification in the NORMA language contains full specification of an algorithm and doesn’t reflect any peculiarities of the computer. It may be implemented on the computer with any architecture both sequential or parallel. Synthesising translator must be automatically adapted to the peculiarities of the architecture or must allow for the peculiar architecture.

As the user is free from the necessity of making a program (this part is carried out by a translator) then such process of programming doesn’t cause mistakes (up to the reliability of a translator). Further more besides the traditional syntactical and semantic diagnostics the synthesising translator can send messages about the errors in the essential notions. E.g., impossibility of organising computation caused by insufficient initial data, by the mistakes in the index displacements etc.

High efficiency of automatically designed program is based on the capability of deep parallelising and providing the necessary level of parallelism granularity. Generally speaking the synthesising translator has the capability of estimating the different variants of the representation of the declarative specification in the program and finding of the best one according to the built-in rule or in the dialogue with the system programmer or the user himself.

Formal semantics’ specification of the NORMA language versions 1-22 are given in this part. Semantics’ specification is based on the application of operational approach and formal methods of relational algebra. References to syntactical notations introduced in specification of the NORMA language by extended notation of Backus-Naur is used here.

A Norma to pJava translation system is among the first software products planned for this part of the pNet project. Not only do program analysis routines for Norma have much in common with those for pJava, but technical developments for pJava directly will make faster running Norma target codes for a wide range of computer architectures.
Translations of Recent Norma Papers From Russian

The last three reports are translations from Russian to English of recent papers in the development of the Norma programming language and its underlying program-analysis and -compilation systems. There are also four older Norma papers still only in Russian.

NORMA - Language specification - Draft copy 1.22

This is the latest version of the formal specifications of Norma language syntax, entitled: "NORMA - Language specification - Draft copy 1.22". It is a 1996 revision of the earlier 50 page preprint number120 “The specification of the NORMA language. Draft Standard,” published at the Keldysh Institute of Applied Mathematics in Moscow in 1995 by the same authors: Alexander Nikolaevich Andrianov, Alexander Borisovich Bugerya, Kyrill Nikolaevich Efimkin, and Igor Borisovich Zadykhailo. It gives the form of all statements in the Norma language.

The NORMA language is a tool aimed at automatic solution of the mathematical physics problems on parallel computer systems. The aim of the NORMA language is to eliminate the programming phase which is necessary to pass from computational formulae derived by an application specialist to a computer program. There is no essential difference between computational formulae and NORMA program structures - these formulae are an input for the NORMA translating system. In fact NORMA program is a nonprocedural specification of problems to be solved. The mathematical problems connected with the synthesis of output program are solvable in the case of the NORMA language. Draft specification of the NORMA language is given.

Organization of loop computations in NORMA language

This is an slightly older paper by Alexander N. (Sasha) Andrianov about the compilation methods to use in determining the execution order of operations in Norma loops: “Organization of Loop Computations in the NORMA Language.” It was preprint number 171 printed in Moscow in 1986 by the Keldysh Institute of Applied Mathematics. The last of the three papers is a continuation of this report, which was 26 pages in Russian.

The problems of loop process organization for the program written in nonprocedural language NORMA are considered in this paper. An algorithm of designing the system of simple loops allowing parallel processing is given. The algorithm is based on the notion of computation’s front which is a hyperplane where variables’ values may be computed in every its point. The task of Linear Integer Programming is solved for determination of the hyperplane’s parameters.

Key words: nonprocedural language, synthesis of the program, parallel computations.
Organization of loop process on specification

This paper “Organization of loop process on nonprocedural specification” by A.N. Andrianov and E.A. Andrianova is a continuation of the 1986 on organization of loop operations in Norma, number 2 in this list of translated papers. It was completed in Moscow in 1996.

NORMA is a programming language [1] aimed at automation of mathematical physics problems solutions on parallel computer systems.

The NORMA language allows elimination of a programming phase in transition from formulae specified by a technical expert to a program itself. There is no much difference between formulae and NORMA specifications. In fact these formulae are input data for a translator.

Synthesis of output program is carried out automatically during the translation from NORMA. The order and the way of performing calculations (parallel, vector or sequential) is determined automatically. The order of the language’s sentences is arbitrary (information dependencies are revealed and taken into account during the organization of computing process). There are no such programming terms as memory, loop, control operators in the language. Output program is generated with the architecture of a target computer as a guide.

In fact the program in NORMA is a nonprocedural specification of the problem to be solved. The synthesis of output program raises some mathematical problems but they are solvable in the case of NORMA language.

Some Norma peculiarities makes the process of automatic object program design available for practice realization. They are:

1. Index expressions of calculated variables has the form \( i = c \) where \( i \) - index name, \( c \) - integer constant.
2. NORMA is a language with single assignment. Any value can be assigned to a variable only once (only once to each point of domain - to the variables defined on domain). The first constraint defines the class of formulae which can be used for the problem’s solution. It isn’t strict in practice as the index expressions of other type are very rare.

Memory allocation and the problems of its economy caused the second constraint. These problems can be solved at the translation stage. The second constraint simplifies the problem of output program synthesis.

The problem of output program is to be solved during the translation. Solving this problem is based on the analysis of the graph of information dependencies. The Most Strongly Connected subGraphs (MSCG) are chosen from the graph. In general case organization of computations for the nodes requires use of special methods.

Assume is a principal operator in NORMA. This operator sets the relations between variables being calculated on a domain. Researching on the subject of computational process organization has been doing for a long time. The purpose of this paper is to specify the method of designing loop operators which realizes the relations included in MSCG (Most Strongly Connected subGraphs).

What we are discussing is an amalgamation of program understanding and transformation technologies to let computers, without help from ultra-skilled humans, write efficient working programs in the future world. Our computers are becoming powerful enough to understand, to manipulate and to improve their own programs, at least for most mundane applications. They are becoming complex enough that soon only
they will be able to write efficient codes for the many variants of parallel and
distributed computing systems. Humans should only point the way to new algorithms
for new problems. Computers should handle all details of producing fast running codes.

The short eight months of research supported by this grant have produced many
sound scientific results. This work points the way to a grand unification of techniques in
program transformation, parallelization, and compilation that will allow the creation of
libraries of reusable portable parallel codes that will run efficiently on almost any
computer system to be found in or on the computer networks of the world.

Appendix:

1) Program Transformations for Java 28 pp.
3) Creation of Efficient and Portable Parallel Programs 54 pp.
4) NORMA - Language specification - Draft copy 1.22 46 pp.
5) Organization of Loop Computations in the NORMA Language 25 pp.
6) Organization of loop process on nonprocedural specification 19 pp.
Program Transformations for Java

Andrei V. Klimov
Keldysh Institute for Applied Mathematics, Moscow

Valentin F. Turchin
The City College of New York

Larry D. Wittie
SUNY at Stony Brook, New York

Abstract

A program transformation system for Java is presented. Two disciplines are reviewed: supercompilation and partial evaluation. Supercompilation is more general and powerful; partial evaluation is a subset, but simpler to understand and to use. Transformation methods were originally developed for functional languages. Here, for the first time, is a demonstration of supercompilation for the imperative language Java. We show how the main phases of a supercompiler work for Java: configuration, driving, configuration analysis. Supercompilation of Java is explained via a Producer-Consumer example, including a trace of manual supercompilation steps.

The strong points of this report are:

It gives an excellent review of program transformation methods.

It shows that the syntax of the popular Java programming language can be used to write computer-understandable portable parallel programs that automatically can be deeply analyzed, optimized and tailored to run efficiently for many specific problems or computer system architectures; and

Its major scientific contribution is a demonstration that supercompilation techniques are already powerful enough to fuse several concurrent threads into one, dramatically increasing the efficiency of some computer programs.

Supercompilers can perform significant interthread analysis and source code optimization. We explain why and how we plan to develop and implement a prototype supercompiler for Java.

This research has been supported by the Office of Naval Research: grant 00014-96-1-0800.
# Contents

**OVERVIEW OF PROGRAM TRANSFORMATION METHODS FOR JAVA** ............................................... 3

**INTRODUCTION** ................................................................. 3

Why supercompilation? ....................................................... 3

Why Java? ........................................................................ 3

**WHAT IS SUPERCOMPILATION?** ......................................................... 4

The notion of a configuration .................................................. 4

Driving ............................................................................... 4

Configuration analysis .......................................................... 5

**WHAT IS PARTIAL EVALUATION?** .................................................. 6

Binding time analysis .............................................................. 6

Partial evaluation proper ....................................................... 7

Partial evaluation as a case of supercompilation ....................... 7

Benefits and limitations of partial evaluation .......................... 7

**COMPOSITE SYSTEM OF PARTIAL EVALUATION AND SUPERCOMPILATION** .................. 8

**CONFIGURATION REPRESENTATION** ................................................. 9

Configuration ........................................................................ 10

Threads ............................................................................. 10

Objects .............................................................................. 10

Configuration variables ......................................................... 11

Restrictions .......................................................................... 11

**PRODUCER-CONSUMER EXAMPLE** .................................................. 12

Introduction .......................................................................... 12

Example ............................................................................ 12

Class Queue .......................................................................... 12

Class Producer ..................................................................... 12

Class Consumer .................................................................... 13

The class ProduceConsume .................................................... 14

1\textsuperscript{ST} VERSION: STATIC LOOP PARAMETER; PRODUCER STARTS BEFORE CONSUMER... 14

Class ProduceConsume ........................................................ 14

Expected result of supercompilation ....................................... 15

2\textsuperscript{ND} VERSION: STATIC LOOP PARAMETER; CONSUMER STARTS BEFORE PRODUCER... 15

Class ProduceConsume ........................................................ 15

Expected result of supercompilation ....................................... 16

3\textsuperscript{RD} VERSION: DYNAMIC LOOP PARAMETER; CONSUMER STARTS BEFORE PRODUCER... 16

Class ProduceConsume ........................................................ 16

Expected result of supercompilation ....................................... 16

**PRODUCER-CONSUMER EXAMPLE: SUPERCOMPILATION TRACE** .......................... 17

Class ProduceConsume ........................................................ 17

Initial configuration ............................................................. 17

Configuration at the end of the Producer constructor .............. 17

Basic configuration: Consumer and Producer to check loop condition \(1 <\leq k\) ........................................ 19

Configuration when Consumer waiting and Producer at start of loop body ........................................ 20

Configuration when Producer notifies .................................... 22

Configuration when Consumer to check loop condition for the second time .......................................... 23

Configuration when Consumer and Producer start their loops again .................................................... 24

**CONCLUSIONS** ..................................................................... 26

**REFERENCES** ..................................................................... 28
Overview of Program Transformation Methods for Java

Introduction

Why supercompilation?

Among the variety of program transformation techniques, two methods perform especially deep optimization:

- partial evaluation [Jones et al 93], and
- supercompilation [Turchin 86, 88, 93, 96].

Originally, these were developed for pure functional languages, but now have become mature enough to be applied to wide-spread practical languages like Java. Partial evaluation is simpler than supercompilation and is better studied. The former is actually a case of the latter. A year ago, in our proposal to ONR, we planned to start by constructing a partial evaluator for Java, and to turn to supercompilation only when we saw that the simpler method turned out well and we understood all subtleties in the transformation of Java programs. However, our research in 1996 has shown that we were too timorous. Hence, this report centers around supercompilation. Our main contribution is that we show how the problem of fusion of several threads into one can be solved by supercompilation. One of the reasons why we have laid partial evaluation aside is that it cannot fuse threads, although we do not see any significant problem in applying it to Java.

At the current stage of research, for Java we plan to use only well-established methods of supercompilation, ones which have already been proved for functional languages. Only after we see how these work in prototype implementation, will we consider using advanced techniques. This report presents the results of a redevelopment of basic supercompilation notions with respect to Java. We are pleased to find that the main ideas remain the same, although a lot of details are new.

Why Java?

Success in program transformation disciplines like supercompilation greatly depends on language properties. It is often said that functional languages are better than imperative for such a task. Indeed, partial evaluation applies to constant subexpressions, that is, program parts that belong to a functional subset. However, this is not essential for supercompilation, which models the operational behavior of a program. What actually matters for all kinds of program transformation is the data model of a language rather than control.

The majority of wide-spread languages, from Fortran to C++, are based on the data model of a von Neumann computer. This means that data definition is two-level. First, clear abstract data domains are defined: integers, strings, records, objects, etc. Second, these are mapped into bit sequences and the one-dimensional array of memory cells. A lot of operations, and even some of the data types (e.g. pointers), refer to the second part of the definition: data coerce and allocate/deallocate structures. This two-level data model badly complicates program analysis and transformation, and is a real (probably, the only) obstacle for bringing the achievements of computer science into practice.

The reason why functional and logic languages are better suited for advanced computing techniques is that from the very beginning they avoided the pressures of program efficiency, and accepted a clean one-level data model. The semantics of programs in languages with a one-level data model can be analyzed and manipulated more easily by other computer programs. Human programming skills are not needed for all aspects of the production of efficient code.
Only a few widely-known languages from the imperative family come close to having a good one-level data model: Algol-68, Ada, and Java. And that is all. We don’t mention such languages as Oberon and Modula-3, since these were not widely used). Java is the simplest of the three, but the most interesting, since it actually supports two programming paradigms. It is both object-oriented and has a functional subset.

By saying that a language has a functional subset we imply that

- its data are developed well enough to represent complex values (at least trees) by immutable data structures;
- functions and procedures can return any values.

To satisfy these requirements, the language implementation must use garbage collection.

Experts in functional programming usually put forward a third requirement that higher-order programming is also supported, that is functions may be used as first class data. However, this is not important for program transformation. On the other hand, Java is good with respect to this feature as well, since functions as values are easily modeled by objects. Java satisfies the functional programming requirements best of all wide-spread imperative languages. Indeed, in [Wadler, Odersky 97] it is demonstrated that Java notions are sufficient to extend it syntactically to a fully functional programming language.

**What is supercompilation?**

This subsection gives an overview of the notions and algorithms of supercompilation. These are explained in more detail along with examples in the next section.

In short, a supercompiler executes a program in general terms, with some data replaced by variables, analyses the trace and constructs an equivalent residual program, which is usually much more efficient that the original one.

In more detail, the definition of supercompilation is two-level. The lower level is a potentially infinite process called driving. The upper-level is configuration analysis, a process which supervises the trace of driving, performs various operations on the trace, controls driving, and constructs a finite graph of a residual program, thus, compiles. Hence, the term super-compilation.

**The notion of a configuration**

The central notion and the main object of the supercompiler is a configuration. A configuration is the representation of (a part of) a generalized state of the machine executing the source program. It resembles a state representation in an interpreter, but contains additional features. To represent sets, free variables, referred to as configuration variables, occur in configurations instead of ground values (final values). The details of Java configurations are defined in the next section.

**Driving**

The basic process in supercompilation is driving, that is execution of a source program in terms of configurations.

---

1 In accordance with the partial evaluation tradition, we refer to a result program, its statements and expressions as residual. The act of generating a residual statement or a residual expression is called residualization.
Driving starts with one or more initial configurations supplied by the user. Each initial configuration generates a task to construct a residual procedure. It contains initial configuration variables, which become parameters of the residual procedure. In order just to optimize a procedure, the user gives the initial configuration that represents the call to the procedure with all arguments taking on different configuration variables. To specialize a procedure, the user gives known values to some arguments.

Driving gradually constructs a potentially infinite driving tree (also referred to as a process tree), which represents the set of all traces of source program execution for the set of its initial states covered by an initial configuration. The nodes of the tree correspond to configurations, the arcs to the steps of execution of the source program. The leaves are passive configurations that contain no procedure calls and usually correspond to the return statement yielding a result value.

A step of driving is either transient, or produces a residual statement (a residualization step).

Transient driving is execution of steps as if by an interpreter, as long as configuration variables do not interfere. Transient steps result in a linear segment of the driving tree. No statements are residualized during transient driving.

When an unknown represented by a configuration variable prohibits execution of a source program statement, or a procedure called in the statement is not given, the statement is residualized. The values of program variables, including the initial configuration variables, are substituted into the statement, and the result statement is residualized and stored in the node of the driving tree together with the configuration. For an unconditional residual statement, only one arc proceeds from the node. Otherwise, two or more arcs leave the node. The arcs enter nodes corresponding to possible next configurations.

The next configuration may be narrowed to represent the information revealed by a step. For example, after a test “if \( x_3 = 5 \ldots \)”, where \( x_3 \) is a configuration variable, it is known that \( x_3 \) equals to 5 on the positive branch. The narrowing information is represented by two means. First, by contraction, that is by substituting some values instead of configuration variables (\( x_3 \mapsto 5 \) in our example). Second, by adding the information to restrictions, which are kept together with configurations. In the basic case (which we plan to implement for Java), restrictions are inequalities (for example, \( x_3 \neq 5 \)). More details are given in the next sections.

Configuration analysis

The driving tree may be considered as an infinite representation of the residual program. Indeed, define its interpretation as follows. Forget configurations, keep in nodes only residual statements and lists of configuration variables. Regard configuration variables as program variables. Assign values to the initial configuration variables and start from the initial node. Execute the residual statement kept in the current node. If only one arc starts from the node, go to the next node. If several arcs proceed from the current node, then the residual statement is conditional, and its execution has selected one of the arcs. Go to the corresponding node. And so forth, until a leaf is reached. The value returned by the expression in the leaf is the result of interpretation.

The interpreter would work similarly if given a finite graph rather than a tree. The finite graph would be an eligible result of compilation. So, the only (but the hardest!) problem is to fold the infinite driving tree into a finite graph. This is the task for configuration analysis.

The following operations on the tree can be performed to fold it:

Looping back: If a configuration \( C \) is a subset of one of the previous ones, \( C' \), and can be reduced to it by a substitution \( \mu \), \( C = \mu C' \), a looping back arc (corresponding to a goto statement in the residual program) is added to the graph plus appropriate assignment statements reflecting the substitution \( \mu \).
Generalization: A subtree starting from a node with configuration C can be thrown away and replaced by a subgraph starting with a more general configuration $C^\mu$, such that $C = \mu C^\mu$ for some substitution $\mu$. The residual assignments corresponding to the substitution $\mu$ are put into node C.

The new graph represents an equivalent but less efficient residual program.

Cutting a configuration: A subtree starting from a node with configuration C can be thrown away and replaced by two subgraphs starting with configurations $C_1$ and $C_2$, which are parts of the original configuration C, and C is a kind of composition of $C_1$ and $C_2$. We will not go into details yet.

However, the residual statement corresponding to C is a call to a procedure $C_1$, while $C_2$ is the continuation of C after $C_1$ returns.

These operations are sufficient to fold any infinite driving tree into a finite graph. However, to detect particular configurations to loop back, to generalize or to cut is the hardest problem of supercompilation. As a first approximation, we will use the method described in [Turkyn 88] after modifying it for multiple threads. The idea is that only the structure of threads is used to make decisions. The values bound to program variables are not taken into account. This method guarantees termination of supercompilation. The details are beyond the scope of this report.

**What is partial evaluation?**

Partial evaluation is a simplification of supercompilation in several respects.

1. In supercompilation one residual program points can emerge from several source program points. This property is referred to as polygenetic, versus monogenetic [Romanenko 90]. Partial evaluation processes each procedure separately. Hence, one residual program point corresponds to only one source program point. This is monogenetic program transformation. Several residual procedures can still be produced from one source procedure. In partial evaluation, there is no problem in determining how to cut a configuration. We may say that cutting happens immediately at each procedure call.

2. A partial evaluator executes subexpressions depending only on known values. Cases when unknowns are present in data, but do not interfere, are not considered as executable by partial evaluation and are always residualized. Compared to supercompilation, partial evaluation leaves many more statements for execution, not eliminated at compile time.

3. The decision when and what to generalize is taken in advance, before known arguments are supplied, by a preprocessor that performs a so-called binding time analysis (BTA). The problem of generalization is drastically simplified as well for partial evaluation.

**Binding time analysis**

Partial evaluation is used to specialize procedures. Given a procedure of several arguments, say void P(int x,y,z), one wants to produce a procedure Q with fewer arguments, say z, with others having been bound to known values, say x=1, y=2. The implicit definition of Q is:

```c
void Q(int z) {
    P(1,2,z);
}
void P(int x,y,z) {
    ...
}
```

A partial evaluator generates specialized versions of codes by executing static subexpressions, ones that depend only on parameters with known values. All other expressions are dynamic. Before the
source program is simplified, it is evaluated by _binding time analysis_ (BTA) to separate and annotate static subexpressions. A parameter of a procedure is considered static if it takes static arguments in all procedure calls. BTA iterates its _abstract interpretation_ until a fixed point is reached. When it starts, all parameters of all procedures are hypothetically considered _static_, except the _dynamic_ parameters of the main procedure. Information about being dynamic is propagated along the functional dependencies of the procedures. Whenever an argument in a procedure call is changed to dynamic, the corresponding parameter becomes dynamic as well, and the procedure body is reanalyzed.

In the P:Q example, if procedure P calls itself recursively in such a way that the second argument y depends on the third z, and the first argument x does not depend on y or on z, x is static and y and z are dynamic.

**Partial evaluation proper**

The second phase is the partial evaluation proper, that is the generation of specialized procedures when known values are given. Static parameters of a procedure may take several different values during this process. Respectively, several specialized versions of the same procedure are generated.

The only requirement is that the number of different values of each parameter must be finite. If this does not hold, the process does not finish, the user should stop it after waiting for some time, and study the dump to find an argument that takes infinite number of values, and forcefully declare it dynamic. Then BTA and partial evaluation are repeated.

**Partial evaluation as a case of supercompilation**

The role of the distinction between static and dynamic parameters can be explained in terms of generalization in supercompilation as follows:

- static arguments are _never_ generalized.
- dynamic arguments are _always_ generalized to a configuration variable.

The user can turn static parameters or subexpressions to dynamic by hand, thus controlling generalization. Simplicity is the main benefit and the main limitation of partial evaluation.

**Benefits and limitations of partial evaluation**

If something can be done by a simpler method, it should be done by it. A great discovery by the authors of partial evaluation was that it is sufficient to convert interpreters to compilers, solving a large class of practical problems. The main benefit of partial evaluation is that it allows clear control by a user who is developing an interpreter and allows experiments with specializing it:

- BTA results are simple for a user to read and to understand. They are just comments annotating the source program. A user may increase the level of generalization (residualization) by marking known (static) variables and subexpressions as unknown (dynamic).
- Termination properties are clear to a user: whether evaluation of static subexpressions results in a finite or infinite set of values of procedure arguments.
- Termination is easily controlled by a user: if a procedure argument takes on an infinite set of values, the user should forcefully mark it as dynamic.

The main limitations of partial evaluation are:

- It does not try to generalize complex configurations than BTA did not pick in advance.
- It processes procedures and threads separately and cannot fuse them (monogenetic).
- It evaluates only subexpressions belonging to a functional subset.

**Composite system of partial evaluation and supercompilation**

We have two powerful program transformation methods which can be implemented for Java:

- One is more general, powerful and complex: supercompilation;
- The other is less powerful, but simpler to understand and to use: partial evaluation.

Supercompilation is not just a single method. It is a series of methods, each based on previous ones. However, in the nearest future, we plan to implement a prototype of just the basic supercompiler, discussed in this report. A practical program transformation system should contain program transformers of different power and with different features. It would be advantageous to have both partial evaluation and supercompilation for Java.

In such a dual system, the two program transformers should be used in the following order:

1. A program should be optimized by the simpler one, partial evaluation;
2. The supercompiler should be applied to the result of partial evaluation.

The supercompiler can perform the work of partial evaluator as well, but the latter can do this more efficiently, especially when the subject program is still under development and its author needs to perform a series of experiments to understand how it behaves under program transformation. Having two program transformers for one practical language, we’ll have a unique opportunity to compare these and get invaluable experience in using different techniques for constructing libraries of reusable software.

However, we plan first to concentrate on basic supercompilation. If we have support enough, a Java partial evaluator will be developed by another researcher.
Configuration Representation

A configuration is a generalized state of a machine executing a source program. It resembles a program state representation in a Java interpreter, but contains configuration variables.

We graphically denote configurations as follows.

**Threads:**

<table>
<thead>
<tr>
<th>“thread name”</th>
<th>☀️ or ☀️</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure name</td>
<td>program point</td>
</tr>
<tr>
<td>variable name</td>
<td>variable value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>procedure name</td>
<td>program point</td>
</tr>
<tr>
<td>variable name</td>
<td>variable value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Objects:**

<table>
<thead>
<tr>
<th>class name</th>
<th>☀️ or ☀️</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable name</td>
<td>variable value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(sub)class name</td>
<td></td>
</tr>
<tr>
<td>variable name</td>
<td>variable value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(sub)class name</td>
<td></td>
</tr>
<tr>
<td>variable name</td>
<td>variable value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

configuration variable (optional)

Object with synchronized methods

**Restrictions:** set of inequalities

More formally, and in more detail, the representation of a configuration is defined as follows:

configuration = (set of threads, set of objects, set of restrictions)

thread = (thread name, active ☀️ or passive ☀️, list of stack frames)

stack frame = (procedure name, program point, list of variable bindings)

object = (list of (class name, list of variable bindings) optionally ended by a configuration variable, safe ☐️ or unsafe ☀️)

variable binding = (variable name, variable value)

variable value = ground value or configuration variable

ground value = simple type value or reference to an object

simple type value = integer or floating point number or character or Boolean
Configuration

A configuration is a triple of a set\(^2\) of threads, a set of objects and a set of restrictions.

Threads

A thread consists of a thread name, a Boolean tag, ⊕ or ⊖ marking whether the tread is active or passive, and a stack of procedure calls (a list of stack frames). The thread name is a string passed as an argument to the Thread constructor. We use it as a comment to clarify the example (e.g. “mainThread”, “Producer”, “Consumer”). A thread is passive if the reference to it occurs in an admission or wait queue. Otherwise it is active. Java semantics allow a thread to wait only in one queue. An admission queue lists the threads waiting to enter a synchronized method. The threads that have executed wait() and have not received notify() stand in a wait queue.

A stack frame consists of a procedure name, a program point and a list of bindings that are pairs of a variable name and a variable value. In our examples, the program point is a number or a letter E that can be found in the program text to the right of the statement; it denotes a point that has been just executed. If a procedure is not static, then the first binding is for the variable this. That is, we consider a method call of form object.method(args) as a procedure call of form class.method(object.args), where the method in defined in the class.

Objects

The main contents of an object are variable names and values. A real supercompiler may avoid spending memory for variable names by renaming variables to ordinal numbers. These are organized in a list of subclasses: first, the top class name and its variables, second, the class name of a first inheritor and the variables defined in it, then the second inheritor, and so on. The list of subclasses may be ended by a configuration variable (see below).

An object has an attribute, safe or unsafe, denoted in figures as ☑ and ☒. Its use is as follows. A configuration represents part of a run-time residual program state. Other runtime threads may refer to the objects of a current configuration and may change them as well. Hence, a supercompiler must be able to imply that variable values may change at any moment, that their object is unsafe. However, if it knows that an object is referenced only from the current configuration, it know that its values change only explicitly during driving. Such an object is marked in a configuration as safe. The safe attribute is present only at (super)compile time and takes no space at run-time. In our examples all objects are safe. See the definition of driving for details of keeping the unsafe attribute.

\(^2\) A set is an unordered finite list. Of course, in a computer, sets are represented by lists with some fixed order. By saying “set” we imply that its order does not matter as well as that we must check different orders of corresponding lists while comparing configurations during supercompilation.
Configuration variables

The supercompiler computes with partially unknown data. The substitutes for unknowns are referred to as configuration variables. A configuration variable may occur instead of any variable value in threads and objects. We denote configuration variables by italic identifiers with subscripts: \( n_1, q_3 \) etc. For clarity in examples, we use the same identifier for a configuration variable as the name of the program variable, which supplies the initial value for that configuration variable. A configuration variable may be copied and assigned to any thread or object variable, and may occur in several places. This represents information about the equality of values in different places. If a configuration variable remains instead of a ground value when a value must be known to perform an operation (e.g. in the if construct), an appropriate statement is put into the residual program. The new configuration may be either the same as before, or changed. In particular, a configuration variable may be contracted, that is replaced by an expression which represents information that has become known. See the definition of driving for details.

One more case of an unknown in a Java configuration is the tail of a list of subclasses in the object representation. When an object has been passed to a procedure as a parameter, say \( x \), of a not final class \( C \), it may actually belong either to \( C \), or to some subclass \( C' \) of \( C \). This is not known until \( x \) is tested by \( x.\text{is} C() \) (or some other Java method). Before the test, the uncertainty is represented by a tail configuration variable. After the test, the configuration variable is contracted and (on the positive branch) replaced by a structure corresponding to the subclass \( C' \), and a new tail configuration variable if \( C' \) is not final as well.

Restrictions

Information about configuration variables is propagated by the supercompiler by two means. First, by substitution to reflect an equality: a configuration variable may be replaced by a ground value or another configuration variable. This operation is referred to as contraction of the variable. This happens when it becomes known that a variable equals to a constant or another configuration variable.

In the special case of a tail variable, the replacement is a structure as just described.

The opposite case of inequality is impossible (or rather, inconvenient) to represent by a substitution. Inequalities are collected as restrictions to the set of states represented by a configuration.

In principle, the supercompiler can work and produce meaningful results without keeping restrictions. The result would be less accurate, the residual program would be larger and less efficient, but correct. It is the essence of supercompilation that the ideal result cannot in general be achieved, and from time to time a supercompiler must forget some restrictions, or perform some other kind of generalization, in order to construct a finite residual program.

The class of restrictions to keep with configurations and the algorithm to compare configurations while accounting for restrictions are a matter of heuristic choice by a supercompiler designer. Keeping in mind a particular class of tasks, we plan to implement the following restrictions in the Java supercompiler:

- \( \text{variable} < \text{constant} \)
- \( \text{variable} \leq \text{constant} \)
- \( \text{variable} = \text{constant} \)
- \( \text{variable} \neq \text{class} \)
- \( \text{name} \)
- \( \text{variable} < \text{variable} \)
- \( \text{variable} \leq \text{variable} \)
- \( \text{variable} = \text{variable} \)
Producer-Consumer Example

Introduction

Threads are used in imperative languages like Java not only for load-balancing multiprocessor systems, but for well-structured programs in object oriented paradigms. However, a highly structured program is often less efficient on a monoprocessor or on a system where the number of processors is significantly less than the number of threads in the program. The following example demonstrates that supercompilation can fuse threads and, in particular, transform a well-structured multithreaded program into a single-threaded efficient one.

Example

Consider a traditional example of a consumer and a producer communicating via a queue object. The example is based on one from the "Java Handbook" by Patrick Naughton, Osborn McGraw-Hill, pp.191-194. The program below consists of a class Queue that implements a queue of integers, a class Producer, a class Consumer\(^3\), and a procedure doit that constructs Queue, Producer and Consumer instances. Two new threads start in Producer and Consumer.

Class Queue

The class Queue has 2 operations:

```java
    void put(int n);
    int get();
```

The first operation puts a integer into a queue. The second operation gets the integer from the queue. The queue can hold at most one element.

```java
class Queue {
    int n;
    boolean valueSet = false;
    synchronized void put(int n) {
        if (valueSet)
            try wait(); catch(InterruptedException e);
        this.n = n;
        valueSet = true;
        notify();
    }
    synchronized int get() {
        if (!valueSet)
            try wait(); catch(InterruptedException e);
        valueSet = false;
        notify();
        return n;
    }
}
```

\(^3\) The class Queue is a copy of the class Q on p.194 in all but its name. The classes Producer and Consumer differ slightly from that on p.192 in the procedure run.
This is a general purpose class which may be thought of as belonging to a library. The following two classes are simple examples of a producer and a consumer, which may be thought of as being written by an end user.

Class Producer

A Producer reads k integers by calling a function My.readInt, and puts them into the Queue supplied by the parameter q. It starts a separate thread.

class Producer implements Runnable {
    Queue q;
    int k;
    Producer(Queue q, int k) {
        this.q = q;
        this.k = k;
        new Thread(this, "Producer").start();
    }
    public void run() throws IOException {
        for (int i=1; i<=k; i++) {
            int n = My.readInt();
            q.put(n);
        }
    }
}

Class Consumer

A Consumer takes k integers from the Queue supplied by the parameter q, applies a function My.F to them and outputs the result by calling a function My.writeInt (the class My and the functions My.readInt, My.F, and My.writeInt are not specified at supercompile time). It starts a separate thread.

class Consumer implements Runnable {
    Queue q;
    int k;
    Consumer(Queue q, int k) {
        this.q = q;
        this.k = k;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        for (int i=1; i<=k; i++) {
            int n = q.get();
            int m = My.F(n);
            My.writeInt(m);
        }
    }
}
The class ProduceConsume

The task for the supercompiler is to optimize a procedure `doit` in a class `ProduceConsume`, which demonstrates a particular use of the `Queue`, `Producer` and `Consumer` classes. It supplies the number of integers to produce and consume, k, to `Producer` and `Consumer` instances. Below, 3 versions of the `doit` procedure and respective expected results of supercompilation (computed manually keeping in mind a particular supercompilation strategy) are presented.

The 1st and 2nd versions are substantially the same case: these differ only in the order of the `Producer` and `Consumer` are constructed and the respective threads are started. Implying that the supercompiler uses a particular order of evaluation (driving) of threads (we use "elder threads are evaluated earlier"), the residual programs differ in the order of calls to `My.readInt` and `My.writeInt`. The essence of these versions is that the loop parameter is static, k=4. Hence, the residual program has simple structure: it is linear. The main problem supercompiler must solve here is not to stop too early. Otherwise, the threads would not be fully fused into one as in the expected results of supercompilation shown below. The criteria to continue driving that has been used during manual supercompilation is natural: the loops are controlled by inequalities of form i<=k, which involve no configuration variables, and the difference between the compared integers decreases.

The 3rd version differs essentially: the loop parameter k is dynamic. Hence, the residual program contains a loop on k (the criteria above does not work, since the loop conditions have configuration variables). The main problem supercompiler must solve here is to catch a moment when to construct a basic configuration, that is a point in the residual program to loop back. The algorithm to estimate 2 configurations as being similar and to generalize them is based on the paper [Turchin 88]. The idea is that to estimate similarity only the form of thread stacks is compared, while the values of program variables are ignored. However, all values are used to construct the least general configuration. Manual supercompilation of the 3rd version is shown below.

1st version: Static loop parameter; Producer starts before Consumer

Class ProduceConsume

```java
class ProduceConsume {
    public static void doit() {
        int k = 4;
        Queue q = new Queue();
        new Producer(q, k);
        new Consumer(q, k);
    }
}
```
Expected result of supercompilation

```java
class ProduceConsume {
    public static void doit() {
        int n1 = My.readInt();
        int n2 = My.readInt();
        int n3 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
        int n4 = My.readInt();
        int m2 = My.F(n2);
        My.writeInt(m2);
        int m3 = My.F(n3);
        My.writeInt(m3);
        int m4 = My.F(n3);
        My.writeInt(m4);
    }
}
```

2nd version: Static loop parameter; Consumer starts before Producer

Class ProduceConsume

```java
class ProduceConsume {
    public static void doit() {
        int k = 4;
        Queue q = new Queue();
        new Consumer(q, k);
        new Producer(q, k);
    }
}
```
Expected result of supercompilation

class ProduceConsume {
    public static void doit() {
        int n1 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
        int n2 = My.readInt();
        int m2 = My.F(n2);
        My.writeInt(m2);
        int n3 = My.readInt();
        int m3 = My.F(n3);
        My.writeInt(m3);
        int n4 = My.readInt();
        int m4 = My.F(n4);
        My.writeInt(m4);
    }
}

3rd version: Dynamic loop parameter; Consumer starts before Producer

Class ProduceConsume

class ProduceConsume {
    public static void doit(int k) {
        0
        Queue q = new Queue();
        new Consumer(q, k);
        new Producer(q, k);
        1
        2
        E
    }
}

Expected result of supercompilation

class ProduceConsume {
    public static void doit(int k) {
        for (int i=1; i<=k; i++) {
            int n1 = My.readInt();
            int m1 = My.F(n1);
            My.writeInt(m1);
        }
    }
}

Two interacting threads have been replaced by a single thread. Supercompilation has found one efficient loop that is equivalent to the entire interaction!
Producer-Consumer Example: Supercompilation Trace

In this section the main steps of the supercompilation process for the Producer-Consumer example are presented. See previous section for the text of the Java program. We consider the 3rd version of the example: dynamic loop parameter; Consumer starts before Producer. Here is the reminder of the procedure to be supercompiled:

Class ProduceConsume

```java
class ProduceConsume {
    public static void doit(int k) {
        Queue q = new Queue();
        new Consumer(q, k);
        new Producer(q, k);
    }
}
```

Initial configuration

Supercompiler starts with one (or more) initial configuration(s). If we are interested in optimizing some procedure, the initial configuration represents the call to it with all parameters bound to distinct configuration variables. In our case, this is the call `ProduceConsume.doit(k_1)`, where `k_1` is a configuration variable that stands for an unknown value of the parameter `k`.

<table>
<thead>
<tr>
<th>Threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;main Thread&quot;</td>
</tr>
<tr>
<td>ProduceConsume.doit</td>
</tr>
<tr>
<td>k</td>
</tr>
</tbody>
</table>

Objects: none
Restrictions: none

Configuration at the end of the Producer constructor

While configuration variables do not interfere, the supercompiler performs transient driving steps that execute a program like a common interpreter. For Java, transient driving imitates execution of a generally multithreaded program on a single processor. Some particular order to evaluate threads must be chosen. Any order is acceptable, but the residual program may differ depending on the order. This example uses the order “eldest thread executes first”. When the next step of the eldest thread cannot be executed, either because it has come to a passive state, or because a configuration variable needs a value used to select one branch of a conditional statement, the next younger thread executes, and so on. After transient driving of all threads has completed, a branching in the residual program is generated, and the process goes further along one branch, then another, as needed. Branching details are discussed below. First, we will see how the supercompiler performs its initial transient driving steps, which are equivalent to the steps of an interpreter.
Figure 3 shows the configuration when mainThread has come to the end of the procedure ProducerProducer at program point E, as marked to the right of the program text. The mainThread dies after procedures ProducerProducer and ConsumeProduce doit return. The threads Consumer and Producer have just been created and are standing at the beginning of their respective run() procedures.

### Threads:

<table>
<thead>
<tr>
<th>“mainThread”</th>
<th>“Consumer”</th>
<th>“Producer”</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProduceConsume doit</td>
<td>Consumer.run</td>
<td>Producer.run</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>this</td>
<td>this</td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer.Producer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>this</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Objects:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Consumer</th>
<th>Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>admission queue</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>wait queue</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>k_i</td>
<td></td>
</tr>
</tbody>
</table>

### Restrictions: none

After mainThread dies, the newly eldest thread Consumer starts executing and enters its loop. Its integer variable i is initialized to 1 and the loop condition i<=k must be checked. After substituting the values i=1 and k=k_i, the supercompiler tries to evaluate 1<=k_i and notices that configuration variable k_i does not allow a branch to be chosen, so transient driving stops for thread Consumer. Thread Producer starts executing and stops in a similar state when it tries to check i<=k as 1<=k_i, at the beginning of its loop. The two threads are evaluating different procedures, Consumer.run and Producer.run, and are checking different i and k variables, which just coincidentally are called by the same local names: Their full names are Consumer.run.i and Producer.run.i, Consumer.run this.k and Producer.run this.k. Coincidentally, the is and ks have the same values. 1 and k_i respectively. The supercompiler will notice these coincidences and generate efficient code.
Basic configuration: Consumer and Producer to check loop condition $i \leq k$

Threads:

```
<table>
<thead>
<tr>
<th>&quot;Consumer&quot;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer.run</td>
<td>1</td>
</tr>
<tr>
<td>this</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>$l_1$</td>
</tr>
<tr>
<td>&quot;Producer&quot;</td>
<td></td>
</tr>
<tr>
<td>Producer.run</td>
<td>1</td>
</tr>
<tr>
<td>this</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>$l_1$</td>
</tr>
</tbody>
</table>
```

Objects:

<table>
<thead>
<tr>
<th>Queue</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>admission queue</td>
<td>[]</td>
</tr>
<tr>
<td>wait queue</td>
<td>[]</td>
</tr>
<tr>
<td>n</td>
<td>$n_1$</td>
</tr>
<tr>
<td>valueSet</td>
<td>false</td>
</tr>
</tbody>
</table>

Residual program constructed till now:

```java
public static void doit(int k) {
    // Initial configuration.
    i1 = 1;
    // Basic configuration.
    M1:
    // ... (Dots denote the residual program is not yet constructed here).
}
```

The configuration in Figure 4 corresponds to the start of the loop in the residual program. Configurations that have two or more predecessors in the residual program graph are called basic. They mark the start of new basic blocks, sequential code sequences amenable to simple transient driving and points where variables used in two or more control threads must be made compatible. Constructing basic configurations is the main task of a supercompiler. It is the most complex problem in supercompilation. Basic configurations emerge as a result of loops.

In this example, when the supercompiler first comes to this point, there is no configuration variable $i_1$. The number 1 is the value for each local $i$ variable. The supercompiler continues (details of how this happens are shown below), executes the first iteration of the loops in Consumer and Producer, and comes to a similar configuration, but with constant 2 for each $i$. The supercompiler must stop iterating and generalize two similar configurations. The heart of the problem for generating residual program loops is recognizing similar configurations.

Deciding whether to stop driving after having met similar configurations is the most complex problem of supercompilation. The basic method shown in this example and planned for our Java implementation comes from the paper [Turchin 83]. This method of comparing and generalizing configurations was developed for lazy evaluation rather than for multiple threads run in applicative order. However, with small changes it can also be used for Java threads.
We do not discuss the general case here, just an idea of how it works for this example. Configurations with stack frames that have the same structure but different variable values are considered similar. When the current configuration is similar to a previous one, the supercompiler stops, and constructs the least common generalization of the configurations. It throws away the parts of the residual program graph that start with the old configuration, appends the generalized configuration the remaining program graph, and puts an assignment of new configuration variables just before the new generalized configuration.

Figure 4 shows the least common generalization of two similar configurations which differed in two places where the first had constant 1 instead of variable \(i_1\), the current had constant 2. The assignment \(i_1 = 1\) is added to the residual program in generalizing the old configuration.

**Configuration when Consumer waiting and Producer at start of loop body**  

*Figure 5*

<table>
<thead>
<tr>
<th>Threads:</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Consumer”</td>
</tr>
<tr>
<td>Consumer.run</td>
</tr>
<tr>
<td>this</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>Queue.get</td>
</tr>
<tr>
<td>this</td>
</tr>
</tbody>
</table>

| “Producer” | [φ] |
| --- |
| Producer.run | 1 |
| this | [·] |
| i | \(i_1\) |

<table>
<thead>
<tr>
<th>Object:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
</tr>
<tr>
<td>admission queue</td>
</tr>
<tr>
<td>wait queue</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>valueSet</td>
</tr>
</tbody>
</table>

| Consumer | [·] |
| --- |
| q | [·] |
| k | \(k_1\) |

| Producer | [·] |
| --- |
| q | [·] |
| k | \(k_1\) |

Restrictions: \(\{i_1 \leq k_1\}\)

Residual program:

```java
public static void doit(int k) {
    // Initial configuration.
    i1 = 1;
    // Basic configuration.
    M1: if (i1 \(\leq k_1\)) goto M2;
    // Current configuration: like the basic one but with restriction \(\{i_1 \leq k_1\}\).
    // ...
    // Configuration like the basic one but with restriction \(\{i_1 > k_1\}\).
    M2:
    // ...
}
```

20

Producer-Consumer Example: Superecompilation Trace
Between the basic configuration in Figure 4 and the configuration in Figure 5, the supercompiler reasons as follows. The next step after the basic configuration cannot be executed unambiguously because of configuration variables in the condition $i_1 \leq k_1$. In such a (general) case, driving is performed as follows. The if statement with an unevaluated condition is put into the residual program, and the process of supercompilation continues along one of the branches. When, sooner or later, it has finished, the supercompiler will continue driving of the second branch. In our example, the if statement in the residual program looks as follows:

\[
\text{if} ((i_1 \leq k_1)) \text{ goto M2;} \\
// Positive branch for condition $i_1 \leq k_1$
\]

M2: // Negative branch

At best, configurations on the branches should represent the exact sets of states into which a conditional statement has split a configuration. At worst, driving of each branch may continue from the same configuration. This is only a question of the quality of the residual program. There is a general rule of supercompilation: "At any moment, a configuration may be replaced by a more general one, reducing the quality of the residual program, but preserving correctness."

The planned implementation of a supercompiler for Java will be able to represent the exact splitting of the configuration in this example. The restriction $\{i_1 \leq k_1\}$ is added to the configuration on the positive branch, and $\{i_1 > k_1\}$ on the negative.

The supercompiler continues driving the positive branch and executes the Consumer thread until it has called wait() in the procedure Queue.run and has been put into the wait queue. This moment is shown in Figure 5.

---

4 Although Java has no goto statement, we use it, since it is a natural means to represent an arbitrary graph of a residual program as a plain text. We'll use a postprocessor to translate gotos to legitimate Java constructs. (See discussion below).
Configuration when Producer notifies

Threads:

```
<table>
<thead>
<tr>
<th>“Consumer”</th>
<th>“Producer”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer.run 3</td>
<td>Producer.run 4</td>
</tr>
<tr>
<td>this</td>
<td>this</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>Queue.get 2</td>
<td>Queue.put E</td>
</tr>
<tr>
<td>this</td>
<td>n</td>
</tr>
</tbody>
</table>
```

Objects:

```
<table>
<thead>
<tr>
<th>Queue</th>
<th>Consumer</th>
<th>Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>admission queue</td>
<td>q, k</td>
<td>q, k</td>
</tr>
<tr>
<td>wait queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>n1</td>
<td>k, k1</td>
</tr>
<tr>
<td>valueSet</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
```

Restrictions: \( i_1 \leq k_1 \)

Residual program:

```java
public static void doit(int k) {
    // Initial configuration.
    i1 = 1;
    // Basic configuration.
    M1: if (!((i1<=k1))) goto M2;
    // Configuration like the basic one but with restriction \( i_1 \leq k_1 \).
    int n1 = My.readInt();
    // Current configuration.
    // ...
    // Configuration like the basic one but with restriction \( i_1 > k_1 \).
    M2:
    // ...
}
```

While the Consumer thread is passive, the Producer thread executes. The supercompiler behaves like an interpreter until it meets the statement int n = My.readInt(). The procedure My.readInt is unknown at supercompile time, and we have nothing to do but to represent the value of n by a new configuration variable \( n_1 \), and to generate the statement int n1 = My.readInt().

Then the Producer goes further and stops when it calls notify() and releases the Consumer. The current configuration is shown in Figure 6.
Configuration when Consumer to check loop condition for the second time

Threads:

```
<table>
<thead>
<tr>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Consumer&quot;</td>
</tr>
<tr>
<td>Consumer.run</td>
</tr>
<tr>
<td>this</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Producer&quot;</td>
</tr>
<tr>
<td>Producer.run</td>
</tr>
<tr>
<td>this</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>Queue.put</td>
</tr>
<tr>
<td>this</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>
```

Restrictions: \( \{i_1 \leq k_1\} \)

Objects:

<table>
<thead>
<tr>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
</tr>
<tr>
<td>admission queue</td>
</tr>
<tr>
<td>wait queue</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>valueSet</td>
</tr>
</tbody>
</table>

Residual program:

```
public static void doit(int k) {
    // Initial configuration.
    i1 = 1;
    // Basic configuration.
    M1: if (!i1<=k1) goto M2;
        int n1 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
    // Current configuration.
    // ...
    // Configuration like the basic one but with restriction \( \{i_1 > k_1\} \).
    M2:
    // ...
}
```

After the Consumer thread has been released by notify(), two statements calling the procedures My.F and My.writeInt are added to the residual program, since the procedures are unknown. A new configuration variable \( m_1 \) represents the unknown value of program variable \( m \).

At the end of the Consumer’s loop, the variable \( i = i_1 \) must be incremented by 1. The supercompiler can behave in different ways. At simplest, it applies the general rule: when a statement or an expression cannot be evaluated, a new configuration variable (say \( i_2 \)) is used to represent its result and an assignment to it is added to the residual program:

```
int i2 = i1+1;
```

Then \( i_2 \) becomes the value of the program variable Consumer.run.i.

A smarter supercompiler can postpone generating the residual statement and represent the expression value by a symbolic expression \( i_1+1 \). The class of expressions to be manipulated by the supercompiler...
must be chosen by the designer of supercompiler for a particular language and data types. We plan to take this decision for Java after a series of experiments. For integers, it seems reasonable to support linear expressions, $a \times b$, where $a$ and $b$ are integer constants and $x$ is a configuration variable.

Figure 7 shows the current configuration for the second method: the value of Consumer.run.i is $i_1+1$.

The supercompiler continues transient driving of the thread Consumer, reaches the beginning of the loop for the second time, and stops, since the loop condition $i <= k$ evaluates to $i_1+1 \leq k_1$, which involves configuration variables. The current configuration does not change and is still as in Figure 7.

**Configuration when Consumer and Producer start their loops again**

After the Consumer thread has been suspended by configuration variables, the supercompiler switches to the Producer thread and drives it until it comes to the beginning of the loop. On the way, the program variable Producer.run.i has been incremented and has taken on the value $i_1+1$ just as explained for Consumer.run.i.

When both loops restart, the current configuration looks like the basic one in Figure 4 except for the values of two program variables, Consumer.run.i and Producer.run.i, and restrictions. The values of both are $i_1$ in the basic configuration and $i_1+1$ in the current configuration. There is no restriction in the basic configuration, and $\{i_1 \leq k_1\}$ in the current one. To loop a current configuration back to a basic one, the set of states the current one represents must be a subset of the set of states of the basic one. This holds when

1) there exists a substitution mapping variables of the basic configuration to expressions, such that it transforms the threads and objects of the basic configuration to those of the current one, and

2) the set of restrictions of the basic configuration is a subset of the restrictions of the current one.

In the example, the first requirement is satisfied by a substitution $\{i_1 \mapsto i_1+1\}$; the second also holds. The substitution becomes an assignment statement of the residual program:

\[ i_1 = i_1+1; \]

By adding the assignment and the goto M1 to the residual program, the supercompiler completes the positive branch of the loop condition.

---

**Residual program:**

```java
public static void doit(int k) {
    // Initial configuration.
    i1 = 1;
    // Basic configuration.
    M1: if (!((i1 <= k1))) goto M2;
        int n1 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
        i1 = i1+1;
        goto M1;
    // Configuration like the basic one but with restriction \{i_1 > k_1\}.
    M2:
    // ...
}
```

Producer-Consumer Example: Supercompilation Trace

24
The supercompiler returns to the configuration on the negative branch (labeled M2), which was laid aside after generating the if statement, and transiently drives it to the end without adding any new statements to the residual program. Supercompilation ends. The final result is as follows.

Residual program:

```java
class ResidualProgram {
    public static void doit(int k) {
        // Initial configuration.
        i1 = 1;
        // Basic configuration.
        M1: if ((i1 <= k1)) goto M2;
            int n1 = My.readInt();
            int m1 = My.F(n1);
            My.writeInt(m1);
            i1 = i1 + 1;
            goto M1;
        M2:
    }
}
```

Since Java has neither the notion of label, nor goto statement, the residual program must be postprocessed. All gotos should be either folded into available Java control constructs, or procedures with goto should be split into several copies, each goto becoming an auxiliary procedure call. The absence of goto is a serious drawback of a language intended for program transformations. Unfortunately, the authors of Java did not think about such use of Java.

Here is the ultimate residual program after postprocessing.

Residual program:

```java
class ResidualProgram {
    public static void doit(int k1) {
        for (int i1 = 1; i1 <= k1; i1++) {
            int n1 = My.readInt();
            int m1 = My.F(n1);
            My.writeInt(m1);
        }
    }
}
```
Conclusions

The example has demonstrated almost all techniques of supercompilation which we plan to implement in the first version of the Java supercompiler. Most are general methods; a few are specific for Java (loop syntax):

1. The notion of configuration:
   - A configuration is
     - a set of states of the original program, as well as
     - a program point of the residual program;
   - Configuration variables are
     - parameters of a configuration, as well as
     - program variables of a residual program;
   - The configuration structure resembles the representation of program state in the Java interpreter and consists of
     - a set of threads, and
     - a set of objects;
   - Additional elements in configurations but not interpretation structures are
     - configuration variables occurring instead of unknown values or the unknown recursive tail of an object,
     - restrictions, which are predicates on configuration variables,
     - the safe/unsafe tag for objects;
   - Each initial configuration supplied by a user generates a task for supercompilation.

2. The process of driving:
   - transient driving: threads are executed as if by an interpreter whenever configuration variables do not interfere. The results of some operations are evaluated even if they depend on configuration variables and are represented by symbolic expressions.
   - residualization of an unconditional statement: when a statement cannot be executed because it depends upon configuration variables or unknown procedures, a copy of it is put into the residual program. Program variables in the copy are replaced by their values, possibly involving configuration variables. If needed, an unknown result of evaluation of an expression is represented by a new configuration variable if it cannot be represented by a symbolic expression.
   - branching (residualization of a conditional statement): when the condition in an if statement cannot be evaluated to true or false, it is residualized and the process of supercompilation continues along one of the branches, usually the positive one first. When supercompilation of the first branch has completed, the second branch is processed. Other conditional statements, for, while, case, are compiled according their semantics by reduction to if.
   - next configurations) after residualization is (are) at worst the same as the one(s) before. However, any additional information revealed during a step is represented in the configuration, improving the residual program. Driving rules for each Java operator algorithmically define how a configuration is narrowed after a step by one of two means:
     - contraction of configuration variables: replacing these by constants or symbolic expressions involving other or new configuration variables;
• adding an elementary predicate to the set of restrictions of the current configuration. A class of elementary predicates is to be chosen by the author of a supercompiler. For Java, we plan to use inequalities, \( \neq, <, \leq \).

3. Configuration analysis:

• The goal of supercompilation is to construct a finite residual program. To achieve this, a supercompiler analyzes the trace of driving, compares the current configuration with previous ones, and takes the following decisions:
  
  • looping back: when the current configuration is a subset of an old one and can be reduced to it by a substitution, appropriate assignment statements reflecting the reduction and a goto\(^5\) statement are put into the residual program. Additional rules may preclude looping back in order to construct a more efficient specialized program.
  
  • generalization: when the supercompiler finds an old configuration which has threads and objects with the same structure as the current configuration, except for variable values, it throws away the part of the residual program starting from the old configuration, builds the least common generalization of the configurations and makes it next after the old one. The corresponding assignments to the new variables of the generalized configuration are put on the arc from the old to the generalized configuration. Then the supercompiler reconstructs the residual program starting from the generalized configuration. This back-tracking is the main reason that supercompilation is often time-consuming.
  
  • cutting\(^6\) the current or one of the previous configurations into two. This corresponds to a procedure call in the residual program. The two parts are supercompiled separately.

• To determine particular configurations to loop back, to generalize or to cut is the hardest problem of supercompilation. As a first approximation, we will use the method described in [Turchin 88] as modified for multiple threads. Its idea is that only the structure of threads, not the values bound to program variables, is used to make decisions. This method guarantees termination of supercompilation.

4. User control.\(^6\) Even if the supercompiler is very clever, it is just a machine, and the human user can substantially help it to catch basic configurations, thus either improving the residual program, or drastically decreasing supercompilation time. However, this is not a task for the end-user, who knows nothing about supercompilation, and is just a programmer using Java source code libraries to write an application in Java. Authors of libraries of reusable software must think about program transformations. We plan to develop the means to annotate library programs with pragma information for the supercompiler. The supercompiler can be given some kind of description of preferable classes for basic configurations. This may include source program points, which may trigger the supercompiler to trace and compare configurations, or to perform some actions suggested by the user.

5. Libraries of reusable software: Supercompilation, as well as other program transformation techniques, allows new methods for structuring programs, in particular, by active use of interpreters. In order to bring supercompilation into practice, we plan to develop some demo libraries, including one to support the construction of database applications.

\(^{5}\) Postprocessor will be used to fold gotos into the Java loop constructs or to translate to auxiliary function calls.

\(^{6}\) This was not demonstrated by the example.
References

[GIÜck, Klimov 95]

[GIÜck, Klimov 95]

[GIÜck, Klimov 97]

[Jones et al 93]

[Klimov 97]

[Romanenko 90]

[Srensen et al 94]

[Turchin 86]

[Turchin 88]

[Turchin 93]

[Turchin 96]

[Wadler, Odersky 97]
pJava
A Parallel Superset of Java
for Automatic Parallelization

Andrei V. Klimov
Keldysh Institute for Applied Mathematics, Moscow

Larry D. Wittie
SUNY at Stony Brook, New York
Abstract

An extension of Java, called pJava, which allows for automatic parallelization and efficient program transformation is presented. The idea of pJava is to select a Java subset with suitable properties, and then gradually extend it while preserving the properties. The starting point of pJava is a purely functional subset of Java with data limited to immutable objects. Higher-level declarative notions will gradually be added to allow for easy, reliable, implicitly parallel programming. Ultimately mutable objects will be allowed as well, but under a special programming discipline. This discipline can be satisfied at a low-level by a qualified Java programmer, or at a high-level via constructs based on monotone objects, which can be easily applied by less-experienced users.

This research has been supported by the Office of Naval Research via grant 00014-96-1-0800.

Contents

INTRODUCTION .............................................................................................................. 3
  Why Java as the basis? .............................................................................................. 3
  Why not Java? Why a new dialect? ..................................................................... 3
  What is the relation of pJava to Java? ................................................................. 3
  What is the background of our research into automatic parallelization of pJava? 4
  What are the characteristic features of the approach? ........................................ 4

PJAV A LANGUAGE STRUCTURE ............................................................................. 5
  VALUE-ORIENTED LEVELS .................................................................................. 5
    1st level .............................................................................................................. 5
    2nd level ........................................................................................................... 5
    3rd level ............................................................................................................ 7
  OBJECT-ORIENTED LEVEL ................................................................................ 7
    The notion of a monotone class ..................................................................... 7
    An example of a monotone class ................................................................... 7
    Monotone object-oriented programming ......................................................... 8

CONCLUSIONS ........................................................................................................... 9

REFERENCES............................................................................................................. 10
Introduction

We propose to develop a language based on Java for easy parallel programming in networks. We call it pJava. The main reasons and goals for this work are as follows.

Why Java as the basis?

- Java is rapidly becoming a wide-spread language used by millions of programmers. A language that has as many features in common with Java as possible will be easier to learn and to use. For the same reason the Java authors based their work on C.

- Java already has good basic properties for automatic parallelization and program transformation. Its authors have done a good job of taking C and removing its poorly defined parts. In particular, the Java data model is good.

Why not Java? Why a new dialect?

- Java is a low-level language. It was not intended for automatic parallelization. It has an explicit method for parallel programming, that is, the notion of threads.

- The level of pJava will be higher than that of Java. Programs in pJava will be suitable for automatic parallelization, both static (at compile time) as well as dynamic (at run time). The method for parallel programming will be implicit and simpler to port to new computers.

What is the relation of pJava to Java?

- First, we select an almost universal subset of Java, which supports program transformation and is suitable for automatic parallelization. The subset is approximately a single-assignment functional language. It is value-oriented rather than object-oriented. Only immutable objects are allowed to model complex values.

- Second, we extend it with high-level constructs, which do not interfere with the goal of automatic parallelization. Extensions are done in cycles, gradually introducing new notions:
  - Value-oriented:
    - Notions from functional programming, including recursive data definitions of the form $x = f(x)$, the semantics of which requires lazy evaluation.
    - Higher level declarative notions from specific applicative areas, especially based on experience from the Norma project [Zadykhalo et al 96]
  - Object-oriented:
    - Restricted Java classes, which allow automatic parallelization. These are referred to as monotone objects [Klimov 91].
  - Other Java means (e.g. threads and arbitrary non-monotone classes) will not be prohibited. These are regarded as low-level, and not recommended for ordinary users, but allowed (and actually needed!) for experienced programmers. The pJava system will contain an analyzer that will annotate a program by telling what parts of a program belong to "true" higher-level pJava, and what are lower-level. This is not just paying tribute to Java. It is our strong opinion that a language containing only high-level notions would be impractical. Moreover,

---

1 An object is immutable if it does not change after a constructor has initialized it. This is the defining property of a class that can be checked by the pJava system.
the idea of a monotone object substantially requires that infinitely many monotone objects be programmed by low-level means.

What is the background of our research into automatic parallelization of pJava?

- Traditional methods of automatic parallelization based on functional, single assignment, data flow languages.
- The experience in parallelization of higher-level declarative languages (like Norma [Zadykhailo et al 96]).
- Our experience in constructing monotone object classes, which allow for automatic parallelization and dynamic program transformation [Klimov 91].
- Achievements in deep program transformation disciplines such as partial evaluation [Jones et al 93] and supercompilation [Turchin 86], and our results in applying these to Java.

What are the characteristic features of the approach?

- Implicit parallelism of higher-level language notions for most users, and explicit notions for parallel programming only for experienced programmers, creating low-level parts of libraries of reusable software.
- Development of high-level languages, up to declarative, for particular application areas, as an extension of a common language, Java. High-level language notions are much better parallelized and transformed than low-level ones.
- Easy parallel programming by the masses as the main goal. This goal is the reason for our preference of high-level notions. If efficiency and simplicity of use conflict, we prefer to sacrifice some efficiency to retain ease of use. keeping in mind that computer speeds almost double each year.
- Our approach differs from that of languages such as HPFlfortran), HPJava [PCRC 96], which are based on extending a language by explicit low-level parallel methods, intended for experienced programmers.
- Use of modern program transformation techniques like supercompilation and partial evaluation to optimize programs deeply and to support automatic parallelization. A motto: “Language properties required for automatic parallelization and efficient program transformation are the same.” Languages well-suited for one goal are well-suited for the other as well. Hence, our simultaneous development of a language for both goals.
- Optimization of parallel programs not only by parallelization, but by “sequentialization” as well, that is, by fusing threads and parallel processes to form smaller numbers of threads and processes with fewer delays for interactions. We consider pJava a high-level language with highly parallel interpretive semantics. To improve its programs, we either extract this implicit parallelism by converting the interpreter to a compiler and mapping it onto a modern computer architecture, or reduce inefficient concurrency by fusing threads.
- Construction of demo libraries of reusable software (for example, for database applications), making use of our methods. To bring the new methods into practice, the authors will develop programs that demonstrate the value of the methods.
pJava Language Structure

pJava is a superset of Java consisting of two parts:

- A **high-level** part, which contains no explicit parallel notions and is intended for easy programming and automatic parallelization. It is based on a suitable Java subset and then is gradually extended preserving the required properties. Call this "pJava proper".

- A **low-level** part, which is a subset of Java, which contains explicit notions for parallel programming.

Together, the low-level and high-level parts cover all of Java. Actually, the low-level part may be defined as containing all Java notions except those belonging to the high-level part. Hence, the user need not limit himself to pJava proper. A preprocessor will annotate each program telling which parts belong to the high and low levels. Such a preprocessor will not only help users, but also is part of an automatic parallelization tool for high-level parts of programs.

The high-level parts of pJava consist of several layers of notions, each more complex than the previous one and relying on it. The layers of pJava are split into two parts: value-oriented and object-oriented.

**Value-oriented levels**

The key idea of the value-oriented part is that mutable objects are prohibited. The values of data entities cannot change. Although object-oriented Java notions such as classes and inheritance are present at these levels, they are used only to model values, that is, immutable data.

**Definition.** A class and its objects are **immutable** if only its constructor sets, or otherwise changes, the value of its local variables, and local objects are also **immutable**.

**Statement.** There exists a Java program analyzer that can check whether a class is mutable or not.

**Note.** The final type modifier of Java supports the notion of immutable objects, and allows the user to supply additional information for the analyzer.

1st level

First, we select a subset of Java, that approximately corresponds to a functional language, and allows for automatic parallelization.

- **Data:** Simple Java types (numeric, characters, Boolean) and immutable classes. No arrays. Only tree structures can be constructed by these means; not graphs.

- **Control:** Procedures programmed in single-assignment style. No threads. Exceptions are allowed under certain conditions.

2nd level

Second, we change and extend the semantics of the 1st level, preserving the syntax.

**Statement.** The 1st level allows for the following changes from sequential to parallel semantics, preserving the upwards equivalence of the result of computation:

- Each class is modified to be a subclass of a special class IsReady that controls an additional tag for each variable telling whether its value is ready, or not. After a variable has been created, the tag equals not ready. After the variable takes its value for the first time, the tag changes to ready. After this, the value does not change.
- Class IsReady has procedures put and get, which are synchronized (in Java terms): get waits for ready, put sends ready signals after it has assigned a value to a variable.
- Each procedure call is executed by starting a new thread.

Statement. The extension of the semantics is a proper extension:

- If a program returns a result by the sequential semantics, it returns the same result by the parallel semantics.
- If a program returns a result by the parallel semantics, it may execute infinitely or reach deadlock under sequential semantics.

Thus, we come to the first extension of Java, which we consider the 2nd level.

Exploiting this semantics extension, new programs may be written to define complex data by means of recursive equations of the form \( x = f(x) \), expressed textually as an ordinary assignment. For example, consider an immutable class to represent integer lists, IntList:

```java
class IntList {
    ...
}
class IntNil extends IntList {
    IntNil() {};
};
class IntCons extends IntList {
    int head;
    IntList tail;
    IntCons(int head, IntList tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

Then assignments of the following form make sense:

\[
\text{IntList } x = F(x);
\]

For example, an \( F \) which defines \( x \) to be a list of natural numbers from 1 to 100 can be coded as follows:

```java
IntList F(IntList x) {
    return new IntCons(1, x.inc().take(99));
}
```

where the methods inc (increment all integers in a list by 1) and take (take first length elements of a list) are declared in the class IntList:

```java
class IntList {
    IntList inc() {
        return
          this instanceof IntNil ? this :
          new IntCons(((IntCons)this).head+1,
                       ((IntCons)this).tail.inc());
    }
    IntList take(int length) {
        return
          this instanceof IntNil || length==0 ?
            (IntList) new IntNil()
            : new IntCons(((IntCons)this).head,
                          ((IntCons)this).tail.take(length-1));
    }
}
```
The procedures `inc` and `take` can execute in parallel because of the extension of the semantics. Old Java syntax is used. No new syntax notions are needed.

3rd level

Here lies a large world of declarative languages, e.g., functional, logical, and constraint languages. First of all, we will develop a Norma-like extension of Java.

The value-oriented part of `pJava` will not allow construction of arbitrary data structures. The most complex structures that can be represented using tail recursion on immutable objects are trees and their simpler degenerate forms, such as linear lists. This limitation gives us a rational reason to include mutable data in further extensions of `pJava`. Their use is not just a matter of improving efficiency, as in the usual arguments against the use of functional languages. Mutable data are necessary.

Object-oriented level

Fortunately, a large number of classes exist which can be programmed by low-level means using synchronized threads, but preserve all properties essential for program transformation when used at the value-oriented level. These classes (and their objects) are referred to as monotone [Klimov 91].

The notion of a monotone class

Definition. A set of classes are called monotone if any Java program that uses the classes and is written so as to satisfy the restrictions of the value-oriented level meets the following properties:

1. **determinacy**: evaluating procedures in arbitrary order according to parallel semantics gives the same result;

2. **recomputability**: if a procedure is evaluated with the same arguments for a second time, it returns an equivalent result, and produces no new side effects.

The first property means that equivalent parallel codes can exist. It makes automatic parallelization possible. The second property allows reliable computations: if a processor on which a process was scheduled fails, the process may be rescheduled and started from the beginning. Our research into supercompilation has shown that the second property is very important for effective program transformations as well.

An example of a monotone class

The following class defines objects which behave like variables with values that can be assigned only once. It is the simplest monotone example.

```java
class IntVar {
    int value;
    boolean ready = false;
    synchronized void put(int n) throws ContradictoryAssignment {
        if (ready && value != n) throw ContradictoryAssignment;
        value = n;
        ready = true;
        notify();
    }
    synchronized int get(int n) {
        if (!ready) wait();
        return n;
    }
}
```

\[ pJava \]
Monotone object-oriented programming

In a similar way, a library of basic monotone classes can be constructed. These use low-level mechanisms like synchronized and boolean ready. The must be written by a qualified programmer. However, when one defines higher-level objects by using only monotone objects as building blocks and no explicit thread-oriented notions are used, the new objects are automatically monotone. Having libraries of monotone functions will allow complex application programs, which are parallel by construction, to be written by ordinary Java programmers.

Monotone classes allow construction of arbitrary complex data, up to graphs. Our experience has shown that, although monotone objects do not allow arbitrary change of state, reactive and dialogue systems can be programmed using a small number of basic monotone objects that are programmed by low-level means.
Conclusions

We intend to use this sequence of extensions to develop pJava from Java. It will be a functional language extended to include at least write-once semantics for data variables, but will retain the familiar syntax of ordinary Java.

Programs based on the functional subset of pJava will allow program transformation and automatic parallelization techniques to be used to form faster running versions of the code for specific purposes. For example, a fast code for multiplying 100x100 matrices can be derived from a generic library routine for multiplying NxN matrices. The routine can easily be ported for efficient execution on any machine for which a pJava interpretation or compilation system exists.

Writing programs in pJava will also allow graphical composition of pJava library routines to form fast running parallel solutions for the specific application needs of masses of users. Whatever code is produced by composition of standard pJava routines can be optimized to run faster, at least by eliminating the parts of the combined routines that are not used in a particular composite calculation. This need will grow as more and more people want to use the Internet to find fast answers to their own questions.
References

[Zadykhailo et al 96]

[Jones et al 93]

[Klimov 91]

[PCRC 96]

[Turchin 86]

[Wadler, Odersky 97]
The report on the subject
Creation of Efficient and Portable Parallel Programs
(grant ONR N00014-96-1-0800)

Contents

1 Technical background of the NORMA language creation 2
   1.1 First works ................................................. 2
   1.2 The advantages of the approach based on the NORMA language usage ... 4
      1.2.1 Extremely high level of the language. New level of stability .... 4
      1.2.2 Portability ............................................. 4
      1.2.3 New level of programming reliability ........................ 4
      1.2.4 Efficiency of the programs ................................ 4
   1.3 Particularizing the statements on the example .......................... 5
   1.4 Principal notions and constructions of the NORMA language .......... 6

2 Semantics of the NORMA language ........................................ 10
   2.1 Background information ....................................... 10
      2.1.1 Notations ............................................... 10
      2.1.2 Elements of relational algebra ............................. 10
      2.1.3 Principal notions and definitions .......................... 12
   2.2 Relations of abstract program $A(P)$ ................................ 14
   2.3 Enviroment of abstract $AM$ machine ................................ 15
   2.4 Definition of preprocessor $Pre(P)$ ................................ 15
   2.5 Specification of preprocessor's operators ............................ 15
   2.6 Definition of translator $Tr(P)$ ................................ 23
   2.7 Specification of translator's operators ................................ 23
   2.8 The rules of $AM$-machine work .................................. 24
   2.9 The example of $AM$-machine work .................................. 41
   2.10 Definite work of $AM$-machine ................................... 52
   2.11 Notes to semantics' specifications ................................. 53
Technical background of the NORMA language creation.

1.1 The first works.

The first publications appeared in late 50s - early 60s [1,2]. The main idea of the language later named NORMA is very simple. It was an attempt to automate the design of the programs based on the jobs prepared by applied mathematicians from Keldysh Institute of Applied Mathematics for further programming. Usually those jobs were the result of applying numerical methods (more often grid method) to physical problems’ solution. The intention was to create the language of jobs’ specification corresponding to the constructions obtained after mathematical solution of the problem.

At first the authors called such method of specification "parameter record" later they used term "nonprocedural specification" nowadays term "declarative specification" is often in use.

Having analysed the material on this subject the authors came to the conclusion that analysis of index dependencies appeared in the relations between variables when continual equations are discretised would be the fact of great importance in designing program on declarative specification.

Let’s consider briefly the first attempt to analyse such relations. There settled the task to analyse specifications of the following type:

\[
\begin{align*}
1. & \quad X_i = f^1(X_{1-\Delta_{1,i}}, X_{2-\Delta_{1,i}}, \ldots, X_{n-\Delta_{1,i}}) \\
2. & \quad X_i = f^2(X_{1-\Delta_{2,i}}, X_{2-\Delta_{2,i}}, \ldots, X_{n-\Delta_{2,i}}) \\
\vdots & \\
\frac{n}{i} & \quad X_i = f^n(X_{1-\Delta_{n,i}}, X_{2-\Delta_{n,i}}, \ldots, X_{n-\Delta_{n,i}})
\end{align*}
\]

(1)

It should be noted that the problem of parallelism wasn't urgent at that time because there were no proper computers that's why they solved the problem of automatic design of the loop process for computations set by relations (1). One can see that a relation is a construction where the variable with indices is placed in the left part and the function from the variables with indices in the right part. In this particular case the variables have only one index and displacement \(\Delta_{k,l}\), \(k,l = 1,\ldots,n\) is an integer constant. General form of relations will be considered below.

Let's consider that the values of variables \(X_i\), \(p = 1,\ldots,n\) are to be calculated for the value of one-dimensional domain set by integer numbers in the ranges from \(m^0(p)\) to \(m^0(p)\), \(m^1(p) >> m^1(p)\), \(p = 1,\ldots,n\), - boundaries of the calculated values. Let's also assume that all the values to the left of \(m^0(p)\) are known. The last assumption is lies in the fact that a full-connected graph corresponds to the dependencies between the variables. The arcs of the graph are considered to connect the graph's node \(X^p\) with node \(X^q\) if \(X^p\) is in the left part of the relation and \(X^q\) is in the right one. We consider the full-connected graph because it has many difficulties in solving.

Matrix \(H\) with elements \(h'(p,q) = m'(q) - m'(p) + \Delta(p,q)\), \(p,q = 1,\ldots,n\) is the main object the method of solution is based on. The variables with index \(t\) are the current state of the boundaries after step \(t\). The values of the components which have the arguments (i.e. they have been already computed or set as the initial values) are assumed to be calculated at every step. Note that \(h'(p,q)\) is the number of components' values which could be computed for variable \(X^p\) if it is depended only on \(X^q\).

Let's give name "chain" to any sequence \(u_1, u_2, \ldots, u_k\) from \(k\) numbers, \(k \leq n\) where all \(u_k\) are different. The sequence of the following type from the elements of matrix \(H\) may be built.

\[h'(u_1, u_2); h'(u_2, u_3); \ldots; h'(u_{k-1}, u_k); h'(u_k, u_1).\]
Let's name sum of all the elements of chain $z$ weight $p'(z)$ of this chain and variable $S'(z) = p'(z)/k$ reduced weight. It is easy to see that $p'$ and $S'$ for every chain don't depend on the system of the boundaries and determined by the index displacements only:

$$S'(z) = \frac{p'(z)}{k} = \frac{\Delta_{n_1,n_2} + \Delta_{n_2,n_3} + \ldots + \Delta_{n_{k-1},n_k} + \Delta_{n_k,n_1}}{k}$$

Corresponding methods of research in organising the loop process of computations is developed in [2].

Let's generalise the results obtained in [2]. Let the chain with the minimal reduced weight $S_M$ be found (an algorithm of determination of chain $S_M$ is given in [2] and has estimate $cn^3 \log_2 n$, $c$ - constant) then the following theorem is correct:

**Theorem.** If the minimal reduced weight of the chain from (1) $S_M \leq 0$ then the computation is impossible. If $S_M \geq 0$ then the computations may be organised in the way that $S_M \cdot n$ values of the variables on the average could be computed at every step.

Here the term "on the average" is used in the sense of the possibility of computing different numbers of the variables at different steps in the case of non-integer $S_M$ but $np$ values of the variables are to be computed for $k$ sequential steps.

**Corollary 1.** Mere sequential computation appears in the case when $S_M$ is reached on the chain with the length $n$ and weight 1, i.e. in the system with the minimal reduced weight $S_M = 1/n$.

**Corollary 2.** Concurrent computation of every variable's values at step $k$ is possible when $S_M \geq 1$.

**Corollary 3.** If $S_M = 1$, $S_M$ is integer then $S_M$ values of every variable can be computed at every step. The regular computation similar to this one may be organised neglecting the fractional part if $S_M$ isn't integer.

It should be noted that though the results given above follow simply from work [2] they are original and show the influence of index dependencies on the possibilities of parallelising. Authors consider the research done on this grant and obtained results very useful.

Given methods didn't have any influence on the design of the translator from the language of declarative specification as A.N. Andrianov constructs general method of computation process organization (parallel, sequential, parallel-sequential) for the systems of relations in rather general form (multidimensional case for the computational domains with complicated configuration is considered). Theoretical results and program implementation of some algorithms based on these results have been obtained.

One more note should be done. It is clear from the task's statement that we'll consider extracting natural parallelism represented in the relations. We aren't going to use any artificial methods dealt with e.g. transformation of the program. E.g. it is generally known that the elements of some tasks can be reduced to the form:

$$y_i = a_i \cdot y_{i-1} + b_i,$$

where the values of $a_i$ and $b_i$ on grid $[1,n]$ and initial value $y_0$ are assumed to be set. $y_i$ is to be computed on grid $[1,n]$. The computation by our methods will be evidently sequential. Really here $n=1$ and $S_M = 1$ so the only one value will be computed at every step.

But it is well-known that there exists method of parallelising such a relation. Authors never refuse to include such methods and even parts of the programs in other languages into translating system. Further more the problem of including Norma extracts into other languages (we mean FORTRAN versions) should be discussed.

The authors also faced the problem of limiting the capabilities for parallelising (the problem of parallelism granularity) or imposing the constraints on the order of computations (e.g. caused by the requirements to the precision of computations).
To conclude this part we should note that its aim was to report some details about the history of the approach to the NORMA language design and to explain principal notions and problems appearing in the solution process on simple examples.

1.2 The advantages of the approach based on the NORMA language application.

1.2.1 Extremely high level of the language. New level of stability

We have gained much experience in designing complicated program systems and translators, now it is time to create really friendly programming languages. We shall take a decisive step and turn from universal languages to the languages for users to formulate problem solution in generic terms. We are sure that universal languages may be friendly only to system programmer. Hence we shall bend every effort to creation of specialized language for each application domain.

It is clear that such a language will be changed as soon as the application domain is changed. So extreme level of the language provides new level of the language stability. Let’s note that the languages of universal type are constantly being specified to get a new level of conveniences in programming or to consider new computer resources. Constant modernisation of the FORTRAN language is a typical example. The idea of creating specialized languages has been known for a long time but it is time to pay much more attention to this approach. We consider this subject to become first and foremost in the programming.

We know by experience that terminology (languages) used by technical experts in the complex application domains gives us a good opportunity to consider the peculiarities of computational environment. The systems of notions in the application domain has been always based on the mathematical notions and it explains this phenomenon. It is worth to mention that there is no notion “memory” in such languages hence there is a principle of single assignment.

1.2.2 Portability

As the specification in the NORMA language contains full specification of an algorithm and doesn’t reflect any peculiarities of the computer, it may be implemented on the computer with any architecture both sequential or parallel. Synthesising translator must be automatically adapted to the peculiarities of the architecture or must allow for the peculiar architecture.

1.2.3 New level of programming reliability

As the user is free from the necessity of making a program (this part is carried out by a translator) then such process of programming doesn’t cause mistakes (up to the reliability of a translator). Further more besides the traditional syntactical and semantic diagnostics the synthesising translator can send messages about the errors in the essential notions. E.g., impossibility of organising computation caused by insufficient initial data, by the mistakes in the index displacements etc.

1.2.4 Efficiency of the programs

High efficiency of automatically designed program is based on the capability of deep parallelising and providing the necessary level of parallelism granularity. Generally speaking the synthesising translator has the capability of estimating the different variants of the representation of the declarative specification in the program and finding of the best one according to the built-in rule or in the dialogue with the system programmer or the user himself.
1.3 Particularising the statements on the example.

There are some characteristics of the language only in connection with the example given below will be briefly described. More details about the language and the state of work are given in the next part. The example given below is taken from the article [4] where the author, Lesly Lamport uses this example to explain the method of hyperplanes applied to the automated parallelising of loops:

\[
\begin{align*}
&\text{DO 99 I=1,L} \\
&\text{DO 99 J=2,M} \\
&\text{DO 99 K=2,N} \\
&U(J,K)=(U(J+1,K)+U(J,K+1)+U(J-1,K)+U(J,K-1))*0.25 \\
&\text{99 CONTINUE}
\end{align*}
\]

This extract is a simplified variant of difference scheme for solving the task of Laplace on rectangular grid \((J,K)\) by fixed number of iteration on \(I\). The constraints imposed by this construction on the computations' order require the analysis of the computations' order (which is set by the nesting of loops), the analysis of variables' values (which is necessary because of the reassignment occurred in carrying out the program), and also the analysis of index dependencies. All these facts must be taken into consideration for the equal results both in parallel and sequential variants.

In the result of the analysis we can state that the values of the variables having index displacements \(J-1\) and \(K-1\) are taken from iteration \(I\) and the rest ones from iteration \(I+1\) in the computation of the values in some point of 3-dimensional space with coordinates \((I,J,K)\) (though only plane \((J,K)\) is set explicitly, variable \(I\) introduces one more coordinate implicitly). All the values of \(U\) are considered to be known on all the boundaries and independent of \(I\). Thus at the first step only the values of variable \(U(2,2)\) when \(I = 1\) can be calculated; at the second step - \(U(2,3)\) and \(U(3,2)\) when \(I = 1\); at the third - \(U(2,4)\), \(U(4,2)\), \(U(3,3)\) when \(I = 1\) and \(U(2,2)\) when \(I = 2\). They proved that there can be found the family of the planes where the values of variable \(U\) in all the points belonged to the next plane in parallel and independent way can be calculated at every step (details see in the article by Lamport).

Note that this extract may be represented in the following form:

\[
\begin{align*}
&U(J,K,I)=(U(J+1,K,I-1)+U(J,K+1,I-1) \\
&\quad +U(J-1,K,I)+U(J,K-1,I))*0.25 \\
&\quad J=1,...,M; \quad K=2,...,N; \quad I=1,...,L
\end{align*}
\]

Up to the notations this variant is a generic mathematical specification where iteration index is consider in a natural way. Besides the specification doesn't contain reassignments of the values to the variables (reuse of memory). So in extracting parallel actions on the specification (3) it isn't necessary to take into consideration neither this fact (we deal with the language with single assignment) nor the order of computing loop iteration. Only the problem of analysis of index dependencies is left.

As in such a description the mathematician sees the geometrical figure - parallelepiped then he mustn't forget the to set known values on the boundaries (the faces of parallelepiped) required for the computation or he will take care of their computation. Even if he has forgotten about them then the message about the error may be sent in the essential terms with the reference to the corresponding face.

Now let's consider the simplest variant of the task's solution scheme:

\[
\begin{align*}
&U(J,K,I)=(U(J+1,K,I-1)+U(J,K,I-1) \\
&\quad +U(J-1,K,I-1)+U(J,K-1,I-1))*0.25 \\
&\quad J=2,...,M; \quad K=2,...,N; \quad I=1,...,L
\end{align*}
\]

This variant is obtained in natural way when the values at a new iteration is obtained by the values at the previous iteration. This variant allows wider and evident parallelising (all the points at a new iteration level may be calculated simultaneously).
The second variant complicates finding the parallelism and decreases the available level of parallelising. This variant of specification may be explained either by the intention of algorithm optimization by means of saving the memory or by the wish of decreasing the number of iteration or by both. This approach is acceptable for sequential realization. But loosing the capability for parallelising may be fatal in the parallel processing environment. Hence the idea of systematical transit from the sequential program to the parallel one (e.g. in HPF [5]) doesn't always produce the acceptable results. On the contrary we consider systematical transit from the specification allowing maximum parallelising to the variants optimising the necessary characteristics by decreasing the level of parallelising rather promising approach. Note that algorithm for parallelising for form (3) may give (in the case of memory optimization) the same level of parallelising and memory capacity as the method of hyperplanes by Lamport for extract (2).

Nowadays there are adherents of the systematical transit from the sequential programs to parallel ones. It refers to some extent to the supporters of HPF approach mentioned above. But there are some opponents of such a method. Let's cite the extract from the article by P.H. Welch, G.R.R. Justo [6]:

- design standards that exclude parallelism also exclude security for complex applications. This leads to growing losses - both financial and human life;
- efficient and robust systems cannot be built by "first getting them to work serially on one processor" and then "parallelising" them;
- existing "dusty-deck" codes that represent massive financial investments that "cannot afford to be wasted", also represent massive serial codes that are becoming unmaintainable and are certainly unverifiable. These are technical dead-ends - as commercial pressures will gradually make clear to all those who persist with them;
- tools to assist the parallelization of large-scale serial code are very difficult to make, will be very expensive to buy and will not be needed by the time they are half-made to work.

From our point of view instead of transit from the sequential version of the program to the parallel one or direct design of the parallel program there is a more promising third way when the original statement of a problem may be realized both in parallel and sequential variants. This approach is based on the main principle: when formulating a new task don't impose extra constraints - further you may face such an environment where they cause inconvenience and inefficiency. Fortunately mathematical specifications are kept to this principle nearly always.

1.4 Principal notions and constructions of the NORMA language

The NORMA language was originally created for specifying the problems of mathematical physics by difference methods. A particular system of notations is formed in this application domain. It is used by technical expert in writing formulae obtained in the process of working out solution's method.

This system of notations contains both common mathematical notions (e.g. integer, real numbers, vectors, matrices, functions and rows) and the notions typical for the given application domain (e.g. grid, index space, grid function, iteration on time and space).

Some abilities of the NORMA language are demonstrated below. More detailed specification is given in [7].

The notion domain is introduced in the NORMA language for representation of index space. Domain is a complex of integer sets \(\{i_1, \ldots, i_n\}, n > 0\) each of them sets coordinates of the point from \(n\)-dimensional index space. Unique name - index name (name of index space's coordinate axe) is linked to every direction (coordinate axe) of \(n\)-dimensional space. Particular case of the notion domain is grid - logical rectangular domain.

One-dimensional domain is used for setting the range of the points on some coordinate axe of index space. The name of one-dimensional domain, the name of index and the range of changing index values are indicated in the simple case of declaration of one-dimensional domain:

**RegionK:** \((k=1..15)\).

The boundaries of the range may be set by constant expressions, e.g.

**GridMN:** \((i=M+3..2*N)\).
Multidimensional domain is built by operation "\(\times\)" of domains' product. The example of two-dimensional domain's declaration is given below. Two-dimensional domain is obtained by domains AxisK and AxisL product:

Square: (AxisK: (k=1..15) ; AxisL: (l=1..5)).

Modification of a domain includes adding some points, deleting some points or changing the range.

Domain may be conditional and unconditional. Conditional domain consists of the points from index space which number and coordinates may be changed depending on satisfying or failure of satisfying the conditions on domain. Unconditional domain consists of the points from index space which number and coordinates may be determined at the translation stage.

It should be noted that the domain determines the values of index space points' coordinates but not the values of calculated variables in these points.

Scalar variables (scalars) are used in NORMA for representation of simple variables but variables defined on domain are used for representation of vectors, arrays and matrixes. Declaration of the variable sets its type - REAL, INTEGER, or DOUBLE and the variable on domain is connected with the domain indicated in the declaration (i.e. the values of this variable may be computed in every point of this domain). Declarations

VARIABLE First, Last DEFINED ON Square

defines variables First, Last on domain Square: it means that the values may be assigned to these variables in every point of domain Square for \(k = 1,...,15, l = 1,...,5\).

Calculating formulae obtained by technical expert are usually written in the form of relations. E.g. calculating formulae for the solution system of linear equations by method of Gauss-Jordan has the form:

\[
\begin{align*}
  m_{o,j} &= a_{o,j}, & j &= 1,...,N, & i &= 1,...,N; \\
  r_{o,i} &= b_i, & i &= 1,...,N; \\
  m_{t,j} &= m_{o-1,t,j}/m_{t-1,t,t}, & j &= 1,...,N, & t &= 1,...,N; \\
  r_{t,i} &= r_{t-1,t}/m_{t-1,t,t}, & t &= 1,...,N; \\
  m_{t,j} &= m_{t-1,t,j} - m_{t-1,t,i} \cdot m_{t,i,j}, & j &= 1,...,N, & i &= 1,...,t-1,t+1,...,N; & t &= 1,...,N; \\
  r_{t,i} &= r_{t-1,i} - m_{t-1,i,i} \cdot r_{t,i}, & i &= 1,...,t-1,t+1,...,N; & t &= 1,...,N; \\
  x_i &= r_{N,i}, & i &= 1,...,N;
\end{align*}
\]

Extract from the NORMA program is given below:

\[
\begin{align*}
  \text{Ot:}(t=0..n) & . \text{Oj:}(i=1..n) . \text{Oj:}(j=1..n). \\
  \text{Oj:}(\text{Oj:Oj}) & . \text{Oij:}(\text{Ot:Oij}) . \\
  \text{Oti:}(\text{Ot:Oj}) & . \text{Oij:}(\text{Oij:Otij/t=1..n}) . \text{Oti:}(\text{Otij/t=1..n}) . \\
  \text{DOMAIN PARAMETERS} & n=10. \\
  \text{VARIABLE} \ \alpha \ \text{DEFINED ON} \ \text{Oij}. & \text{VARIABLE} \ \mu \ \text{DEFINED ON} \ \text{Oij}. \\
  \text{VARIABLE} \ \beta \ \text{DEFINED ON} \ \text{Oij}. & \text{VARIABLE} \ \gamma \ \text{DEFINED ON} \ \text{Oij}. \\
  \text{INPUT} \ \alpha \ \text{ON} \ \text{Oij}, \ \beta \ \text{ON} \ \text{Oj}. & \text{OUTPUT} \ x \ \text{ON} \ \text{Oj}. \\
  \text{FOR} \ \text{Otij/t=0} \ \text{ASSUME} \ \mu = \alpha.
\end{align*}
\]
FOR Oti/t=0 ASSUME r=b.
OiEQ[i]1, OiNE[i]1;Oti[i]/i=t. OiEQ[i]1, OiNE[i]1;Oti[i]/i=t.
FOR OtiEQ[i]1 ASSUME m = m[i-1,i=t]/m[i-1,i=t,j=t].
FOR OiEQ[i]1 ASSUME r = r[i-1,i=t]/r[i-1,i=t,j=t].
FOR OiNE[i]1 ASSUME m = m[i-1,j=t]m[i,t]=t.m[i]=t.
FOR Oi[i]1 ASSUME r = r[i-1]-m[i-1,j=t]r[i]=t.
FOR Oi ASSUME x = r[i=n].

Necessary computations are described by **ASSUME** operator.

**FOR domain ASSUME relation.**

This operator is a key-notion of the NORMA language. The relation set the role of computing
the variable's value from the left part on the values of the variable from the right part and index
dependencies between the variables. The variable from the left part is to be computed in all the
points of the **domain**; the rule for this computation is defined definitely but immediate fulfilment of the
computation isn't required in the given place of the program and the method and the order of the
computation isn't set. Some formal similarity with assignment operator mustn't mislead you.
Indices without displacements in the formulae notations may be omitted because they are
automatically restored by the translator in analysing the program.

There are also conditional domains used in the considered extract. Thus definition **OiEQ[i],**
**OiNE[i]1;Oti[i]/i=t** determines two disjoint subdomains **OiEQ[i]1** and **OiNE[i]1**. The first
subdomain consists of the points from modified domain **Oti[i]1** where given condition **i=t** is true and
the second of the points where this condition is false (i.e. **i≠t**). In general case the condition in
declaration of conditional domains is represented by logical expression.

The differences in the given above two ways of writing calculating formulae are in the form of
writing (index representation, linearity of the specification) but they are equivalent in their contents.

Special instruction **ITERATION** is used in the NORMA language for the description of
iterative processes.

Declarations of input or output variables are used in their usual way. E.g. declarations

**B1,B2: B/z<ɛ.**
**INPUT Velocity ON A. OUTPUT Tau ON B1.**
**INPUT X, ALFA.**

are the requests for input of the values of scalars **X, ALFA,** variable **Velocity** in all the points of
domain **A** and also for output of the value of variable **Tau** in all the points of domain **B** where
condition **Z<ɛ** is satisfied.

The order of input or output of the variables' values isn't defined, it is determined in the
process of translation of the program automatically (it requires special processing of the data files by
the translator).

To finish the introduction of some NORMA facilities we consider reduction functions. They are
analogies of the traditional mathematical notations of \( \sum , \Pi \) max, min type. Call to these
functions has the form:

\[
\text{name-of-function} \left( \left( \text{name-of-domain} \right) \text{arithm-expression} \right).
\]

**Domain** determines the set of the domain's points where the function is to be computed,
arithmetical expression is a set of the values which the function is to be applied to. Let's compute

\[
V_i = W_i + \sum_{j=1}^{m} A_{i,j} \times X_j \quad i=1,...,n.
\]
The description of this computation in the NORMA language

VARIABLE A DEFINED ON Grid : (Oi:(I=1..N);Oj:(J=1..M)).
VARIABLE V,W DEFINED ON Oi. VARIABLE X DEFINED ON Oj.
FOR Oi ASSUME V = W+sum((Oj) A*X).

coincides (up to the notations) with the original calculating formula. Reduction functions' realization for different computational environments isn't an easy problem but this problem is solved by the translator not the user.
2 Semantics of the NORMA language

Formal semantics' specification of the NORMA language versions 1-22 [7] are given in this part. Semantics' specification is based on the application of operational approach [8,9] and formal methods of relational algebra [10,11].

2.1 Background information

2.1.1 Notations

There are some notations used below:

- \( \mathbf{R} \) - set of real numbers;
- \( \mathbf{N} \) - set of natural numbers;
- \( \mathbf{I} \) - set of integer numbers;
- \( \mathbf{T} \) - set of program \( P \) identifiers;
- \( \mathbf{F} \) - set of program \( P \) arithmetical expressions;
- \( \mathbf{L} \) - set of program \( P \) logical expressions;
- \( \emptyset \) - empty set;
- \( \Lambda \) - value isn't defined;
- \( \cdot \) - operation of strings' concatenation;
- \( S_1 \preceq S_2 \) - relation of occurrence of substring \( S_1 \) into string \( S_2 \);
- \( \|A\| \) - power of set (or relation) \( A \);
- \( \sim \) - "is by definition";
- \( := \) - "make equal";
- \( \mathbf{T}, \mathbf{F} \) - logical values \textbf{TRUE, FALSE}.

The construction of the choice has the form:

\[
(P_1 \rightarrow O_1; P_2 \rightarrow O_2; \ldots ; P_n \rightarrow O_n)
\]

where \( P_1, \ldots , P_n \) - finite number of statements, \( O_1, \ldots , O_n \) - any objects. It determines the first (counting from the left to the right) object \( O_1 \) which has \( P_1 = \mathbf{T} \). Besides

- if all \( P_i = \mathbf{F}, \quad i = 1, \ldots , n \) then the construction of choice is an empty object;
- if the variables included into \( P_i \) are used in setting the object \( O_i \) then the values assigned to these variables have \( P_i = \mathbf{T} \);
- if terms of \( \forall x F(x) \) type are included into \( P_i \) the usage of the variable \( x \) in setting object \( O_i \) means that all the values may be assigned to this variable (in the arbitrary order) when \( F(x) = \mathbf{T} \);
- if terms of type \( \exists x F(x) \) are included into \( P_i \) the usage of the variable \( x \) in setting object \( O_i \) means that any value may be assigned to this variable (but only one) when \( F(x) = \mathbf{T} \).

2.1.2 The elements of relational algebra

Give some definitions according to e.g.[10].

\( D_1, \ldots , D_k \) - some (may be identical) sets.

\( R \) is said to be a relation on sets \( D_1, \ldots , D_k, \quad k \geq 1 \), if \( R \subseteq D_1 \times \ldots \times D_k \) where \times - operation of Cartesian product.

Sets \( D_1, \ldots , D_k \) are called domains of relation \( R \).

Elements \( \langle d_1, \ldots , d_k \rangle \) of relation \( R \) which have \( d_i \in D_i, \quad i = 1, \ldots , k \) are called tuples of relation \( R \).
Specification of type $R(A_1, \ldots, A_n)$ are called scheme of relation $R$ and $A_i$ is an attribute of relation $R$.

Two types of operations are defined on relations: traditional operations on sets (union, intersection, Cartesian product etc.) and special relational operations (choice, filtration, projection, interconnection).

Let's introduce the following notations:
$R_1, R_2, R$ - names of relations,
A - attribute of relation R,
C - logical condition,
$\alpha, \beta$ - tuples of relations.
$R_1 \otimes R_2$ - operator of relations' Cartesian product.
Arguments: relations $R_1$ and $R_2$.
Result: extended Cartesian product of relations $R_1$ and $R_2$. If $R_1 = \langle \alpha_1, \ldots, \alpha_n \rangle$, $R_2 = \langle \beta_1, \ldots, \beta_m \rangle$ then
$$R_1 \otimes R_2 = \alpha_1 \circ \beta_1, \alpha_1 \circ \beta_2, \ldots, \alpha_n \circ \beta_m.$$ 
$R_1 \ AD T O \ R_2$ - operator of adding relation $R_1$ to $R_2$.
Arguments: relations $R_1$ and $R_2$ compatible by union i.e. having equal numbers of attributes but for $i=1,\ldots,n$ the value of one and the same domain $D_i$ is assigned to $i$-attribute of $R_1$ and $R_2$.
Result: relation $R_2$ added with the tuples of $R_1$ (without repetitions).
$R_1 - R_2$ - operator of $R_1$ and $R_2$ relations' difference.
Arguments: relations $R_1$ and $R_2$ compatible by union.
Result: relation $R$ containing the tuples of relation $R_1$ which aren't included into relation $R_2$.
$F(R, C)$ - operator of filtration.
Arguments: relation $R$ and logical condition $C$ built from constants and attributes of relation $R$ by means of comparison operations $=, \neq, >, <, \geq, \leq$ and logical operations $\lor, \land, \neg$.
Result: relation consists of those tuples of relation $R$ where condition $C$ is true.
$R(A)$ - operator of interception.
Arguments: relation $R$ and its attribute $A$.
Result: relation consists of the values of attribute $A$ only in relation $R$ (without repetitions).
$R(\overline{A})$ - operator of projection.
Arguments: relation $R$ and its attribute $A$.
Result: relation, consists from the tuples of relation $R$ where the values of attribute $A$ (without repetitions) are deleted from.
$\alpha \circ \beta$ - operator of tuples' interconnection.
Arguments: $\alpha = \langle a_1, \ldots, a_n \rangle$ and $\beta = \langle b_1, \ldots, b_n \rangle$.
Result: tuple $\alpha \circ \beta = a_1, \ldots, a_n, b_1, \ldots, b_n$.
$R_1 \circ R_2$ - operator of relations' interconnection.
Arguments: $R_1 = \{ \alpha_1, \alpha_2, \ldots, \alpha_n \}$ and $R_2 = \{ \beta_1, \beta_2, \ldots, \beta_m \}$, $R_1$ and $R_2$ - relations with equal power
Result: relation $R = \{ \alpha_1 \beta_1, \alpha_2 \beta_2, \ldots, \alpha_n \beta_m \}$.
$f(R)$ - operator of relations' transformation.
Argument: $R = \{ \alpha_1, \alpha_2, \ldots, \alpha_n \}$
Result: relation $R = \{ f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_n) \}$.
$UPDATE(\alpha, \beta, R)$ - operator of tuple's renewal.
Arguments: relation $R$, tuple $\alpha = \langle a_1, \ldots, a_n \rangle \in R$, tuple $\beta = \langle b_1, \ldots, b_m \rangle$ where the values of the same domain $D_i$ are assigned to $\alpha_i$ and $\beta_i$. 

Result: relation $R$ where tuple $\alpha$ is substituted by tuple $\beta$.

Let's consider arranged relation $\Pi(I), \quad I \in \mathbb{N}$ defined.

### 2.1.3. Principal notions and definitions.

References to syntactical notations introduced in specification of the NORMA language [7] by extended notation of Backus-Naur is used here. Further the grammar of the NORMA language will be denoted as $G_{\text{Norma}}$ and the language engendered by this grammar $L(G_{\text{Norma}})$. Syntactical notations will be typed in italic e.g.: \textit{main-part}, \textit{declaration-of-domain}.

The process of $P \in L(G_{\text{Norma}})$ program computation is come to:

- transformation of $P$ to standard form $P_s$ by sequence $U_1, \ldots, U_n$ of preprocessor's operators;
- design abstract program $A(P)$ on $P_s$, the abstract program is represented in the form of relations \textit{Domains}, \textit{Variables}, \textit{Input}, \textit{Output}, \textit{InputData}, \textit{Relations};
- interpretation of abstract program $A(P)$ by abstract $AM$ machine it is come to carrying out sequence $Q_1, \ldots, Q_k$ operators of the relational algebra and modifying the relations given above.

The process of interpretation consists of the sequence of stages. The definition of interpretation's stage is given below in the description of the rules of $AM$ machine work.

Thus the semantic $\text{Sem}(P)$ of the program $P$ is determined by the following process:

$$\text{Sem}(P): \quad P_s := \text{Pre}(P), \quad A(P) := \text{Tr}(P_s), \quad AM(A(P)),$$

where $\text{Pre}$ is a preprocessor transforming $P$ to equivalent standard form $P_s$;

$\text{Tr}$ is a translator preparing abstract program $A(P)$ for $AM$ machine on standard representation $P_s$. Functions of preprocessor $\text{Pre}$ and translator $\text{Tr}$ are defined in the next parts.

Operators $U_j, \quad j = 1, \ldots, n$ and $Q_i, \quad i = 1, \ldots, k$ may be \textit{conditional} and \textit{unconditional}. The transformation of program $P$ is the result of carrying out unconditional operator $U_j$. Changing of relations' tuples is the result of carrying out conditional operator $Q_i$. Value $T$ or $F$ is the result of carrying out $U_j$ or $Q_i$.

Program $P$ contains semantic error if

$$\exists j: U_j = F \lor \exists i: Q_i = F$$

Program $P$ is \textit{semantically correct} if there is no semantic errors in it.

Variable $X_1$ of program $P$ has \textit{information dependence} on variable $X_2$ of program $P$ if the value of $X_2$ or the value of $X_3$ which has information dependence on $X_2$ is necessary for the computation of $X_1$ value. Variable $X_1$ has \textit{information independence} if there is no variable $X_2$ of program $P$ which variable $X_1$ depended on. The value of $X_1$ is \textit{directly computable} if it has information independence or the values of all the variables $Y_1, \ldots, Y_n$ which $X_1$ is depended on have been already computed.

Abstract $AM$ machine is a non-determined device: semantics $\text{Sem}(P)$ makes correspondence of semantically correct program $P$ to the class of sequences made up by the operators of the relational algebra which performance by $AM$ machine reflects all the possibilities of computations parallelization and gives identical results.

The parallelism determined only by information dependencies between the variables of program $P$ is said to be \textit{natural (ideal) parallelism} of program $P$. Informally natural parallelism means that at every moment the values of all the directly computable variables of program $P$ are being computed. From this point of view semantics $\text{Sem}(P)$ may be considered as \textit{the semantics of natural parallelism} as in particular $AM$ realises the natural parallelism of program $P$. 

Let's define the way of representing syntactically correct program \( P \) (syntactically incorrect programs aren't considered as they don't belong to \( L(G_{\text{Norma}}) \)).

Define set \( E \) of elementary objects, set \( H \) of compound objects and set \( O \) of language \( L(G_{\text{Norma}}) \) objects:

\[
E = \{ \text{identifier, key-word, constant, sign-of-operation, delimiter} \};
\]

\[
H = \{ \text{program, part, declaration, operator, ...}, \text{declaration-of-domain, declaration-of-input,..,arithmetic-expression,..} \}
\]

\[
O = E \cup H
\]

Informally set \( H \) consists of the language constructions engendered by all the non-terminals of grammar \( G_{\text{Norma}} \) except non-terminals included into \( E \) and principal-symbols.

Consider system \((O, S)\), \( S \) is a set of selectors.

Let \( is-\alpha \) be a predicate, defined on \( O \), \( is-\alpha : O \rightarrow \{T,F\} \). Selector of \( is-\alpha \) is said to be subdomain \( O \) satisfied predicate \( is-\alpha \).

E.g.

\[
is-\text{identifier} = \{ x | is-\text{identifier}(x) \}
\]

For the case when selector \( is-\alpha \) contains the only element a let's use notation \( a \) (where it doesn't cause ambiguity). E.g. terminals \( is-( \) and \( is-\text{BEGIN} \) will be represented as \( ( \) and \( \text{BEGIN} \) correspondingly.

Program \( P \) is corresponded to the system \((O_P, S_P)\), \( O_P \subseteq O \) and \( S_P \subseteq S \) and it is represented in the form

\[
P = o_1 \cdot o_2 \cdot ... \cdot o_n = o_1o_2...o_n, \quad n \geq 1, \quad o_i \in O_P
\]  \hspace{1cm} (5)

and the application of selector \( s \in S_P \) to a program (or its part) in the form (2) is represented in the form \( s(P) \) (correspondingly \( s(o) \), \( o \in O_P \)).

Let's introduce some auxiliary operators and functions.

\text{position}(o_i, o) - function which determines the number of subobject \( o_i \) in object \( o \) for representation of \((5)\) type

\[
\text{position}(o_i, o) = i
\]

Object \( o_i \) is said to be to the left of object \( o_j \) in object \( o \) (notation \( o_i <_o o_j \)) if \( \text{position}(o_i, o) < \text{position}(o_j, o) \).

\text{Replace}(o_1, o_2, o) - \text{operator of substitution} \( o_2 \) instead of \( o_1 \) in \( o \); \( o_1, o_2, o \in O_P \).

\[
\text{Replace}(o_1, o_2, o) \sim (o = A \cdot o_1 \cdot B \rightarrow A \cdot o_2 \cdot B)
\]

\text{ReplaceAll}(o_1, o_2, o) - \text{operator of substitution} \( o_2 \) instead of all the occurrences of \( o_1 \) into \( o \); \( o_1, o_2, o \in O_P \).

\[
\text{ReplaceAll}(o_1, o_2, o) \sim (\forall o_1 (o_1 \neq o) \rightarrow \text{Replace}(o_1, o_2, o))
\]

\text{Delete}(o_1, o) - \text{operator of deleting the occurrence of} \( o_1 \) into \( o \); \( o_1, o \in O_P \).

\[
\text{Delete}(o_1, o) \sim (o = A \cdot o_1 \cdot B \rightarrow A \cdot B)
\]

\text{DeleteAll}(o_1, o) - \text{operator of deleting all the occurrences of} \( o_1 \) into \( o \); \( o_1, o \in O_P \).

\[
\text{DeleteAll}(o_1, o) \sim (\forall o_1 (o_1 \neq o) \rightarrow \text{Delete}(o_1, o))
\]

\text{Include}(o_1, o) - \text{operator of including} \( o_1 \) into \( o \), keeping syntactical correctness of object \( o \); \( o_1, o \in O_P \).

\text{iname}(P) - \text{generation of the name unique in program} \( P \).
\(\text{uname}(P) = o, \ o \in \overline{\text{id-identifier}(O)} \land \neg \exists p(p \in O \land o = p)\).

2.2 Relations of abstract program \(A(P)\).

**Domains** (Cond, D, Iname, I): domains of program \(A(P)\).
- \(\text{Cond} \in L - \text{logical expression.}\)
- \(D \in T - \text{identifier of domain.}\)
- \(\text{Iname} = I_{i} \circ \ldots \circ I_{n}, \ I_{i} \in T - \text{identifier of domain's indices, } n - \text{dimension of domain } D.\)
- \(I \in D_{1} \circ \ldots \circ D_{n}, \ D_{i} \in N - \text{index of domain } D.\) The values of \(D\) domain (sets of integer numbers) are values \(I.\)

Tuple \(\langle \text{Cond} \circ D \circ \text{Iname} \circ I \rangle \) belongs to relation **Domains**, if point \(i_{1}, \ldots, i_{n}\) of index space \(I_{1} \circ \ldots \circ I_{n}\) belongs to domain \(D.\)

**Variables** (D, X, Iter): definition domains of program \(A(P)\) variables.
- \(D \in T \cup \emptyset - \text{identifier of domain.}\)
- \(X \in T - \text{identifier of variable.}\)
- \(\text{Iter} \in T - \text{identifier of iteration index.}\)

Tuple \(\langle D \circ X \circ \Lambda \rangle\) belongs to relation **Variables** if variable \(X\) is defined on domain \(D.\) If variable \(X\) is indicated in the list of iterated variables with iteration index \(\text{Iter}\) then tuple of relation **Variables** has the form \(\langle D \circ X \circ \text{Iter} \rangle\).

**Input** (Var, Value, File): input variables of program \(A(P)\).
- \(\text{Var} = X(I), \ X \in T - \text{identifier of an input variable, } I \in D_{1} \circ \ldots \circ D_{n}, \ D_{i} \in N - \text{index of input domain } D\) (set of integer numbers).
- \(\text{Value} \in R \cup \Lambda - \text{values of input variable } X \text{ with indices } I.\)
- \(\text{File} - \text{name of an input file.}\)

**Output** (Cond, Var, Value, File): output variables of program \(A(P)\).
- \(\text{Cond} \in L - \text{logical expression.}\)
- \(\text{Var} = X(I), \ X \in T - \text{identifier of an output variable, } I \in D_{1} \circ \ldots \circ D_{n}, \ D_{i} \in N - \text{index of output domain } D\) (set of integer numbers).
- \(\text{Value} \in R \cup \Lambda - \text{values of output variable } X \text{ with indices } I.\)
- \(\text{File} - \text{name of an output file.}\)

**InputData** (Var, Value, File): the values of input variables of program \(A(P)\), stored in the external files.

Attributes are coincided with the attributes of relation **Input** except the set of values of attribute Value: Value \(\in R.\)

**Relations** (Name, InVal, Def, Cond, Var, Value, Func): computed variables of program \(A(P)\).
- \(\text{Name} \in T - \text{identifier of } A(P) \text{ program's part.}\)
- \(\text{InVal} \in \{\text{ItId} \circ \text{C}\} \cup \{\emptyset, \emptyset\} - \text{character of computation in construction iteration} \ (\text{InVal} = \{\text{ItId} \circ \text{C}\}) \text{ or outside iteration } \text{InVal} = \{\emptyset, \emptyset\}; \ \text{ItId} \in T - \text{identifier of iteration index.}\)
- \(\text{C} \in 1 - \text{the value of iteration index.}\)
- \(\text{Def} \in L - \text{logical expression.}\)
- \(\text{Cond} \in L - \text{logical expression.}\)
- \(\text{Var} = X_{1}(I_{1}) \ldots X_{k}(I_{k}), \ k \geq 1 - \text{variables computed in part Name. } X_{i} \in T \ i = 1, \ldots, k - \text{identifiers of variables; } I_{i} \in D_{1} \circ \ldots \circ D_{n}, \ i = 1, \ldots, k, \ D_{i} \in N - \text{the values of computation's domain } D_{1} \circ \ldots \circ D_{n} \text{ (sets of integer numbers).}\)
\[ Value = Value_1, \ldots, Value_k; \quad Value_i \in \mathbb{R} \cup \Lambda \quad i = 1, \ldots, k - \text{values of computed variables} \]
\[ Var = X_1(I_1) \ldots X_k(I_k), \quad k \geq 1. \]
\[ Func \in F - \text{type of functional dependence for variables } Var. \]
The tuples in every relation defined above are considered arranged in some fixed way.

2.3. Environment of abstract AM machine.

Environment of abstract AM machine
\[ Env = \{ \text{Domains, Variables, Input, Output, Relations, InputData, IndSpace, Files} \} \]
\[ IndSpace = \{ IndSpace_i \}, \quad i = 1, \ldots, k, \quad k - \text{number of program } P \text{ parts}; \]
\[ IndSpace_i = \text{name-part}_i \cdot o'_1 \cdot o'_2 \cdot \ldots \cdot o'_n, \quad n \geq 0, \]
\[ o'_j \in is-name-index(part_i), \]
\[ Files - \text{the names of external data files.} \]
\[ Files = \{ \text{filename}_1, \ldots, \text{filename}_l \}, \quad l \geq 0. \]

2.4 Definition of preprocessor Pre(P).

Preprocessor Pre translates the text of some program P into equivalent standard representation P, or fixes semantic errors.

\[ Pre(P) \sim \text{MAIN-EXIST}(P); \]
\[ \quad \left( \forall o(o \in is-part(P)) \rightarrow \right. \]
\[ \quad \text{LOCAL-VARIABLES-RENAME}(o); \]
\[ \quad \text{NONREC-DOMAIN-DECLARATION}(o); \]
\[ \quad \text{EXIST-UNIQ-DECLARATION}(o); \]
\[ \quad \text{MACRO-INDEX-EXCLUDE}(o); \]
\[ \quad \text{DEF-INDEX-SPACE}(o); \]
\[ \quad \text{PAR-DOMAIN-EXCLUDE}(o); \]
\[ \quad \text{CONST-EXPR-EVALUATE}(o); \]
\[ \quad \text{STANDARD-DOMAIN}(o); \]
\[ \quad \text{STANDARD-ASSUME}(o) \right). \]

2.5 Specification of preprocessor's operators.

\( \diamond \) Checking main part existence MAIN-EXIST(P).
Main part must be in the program and be the only one.

\[ \text{MAIN-EXIST}(P) \sim \exists! o(o \in is-main-part(P)) \]

\( \blacklozenge \) Renaming local variables LOCAL-VARIABLES-RENAME(o).
Names localized in the parts are substituted to the unique ones. The names of parts and functions aren't renamed. Localized in the parts names of input variables (scalars and variables defined on domain) are renamed according to the names of input variables set in the input files.

\[ \text{LOCAL-VARIABLES-RENAME}(o) \sim \]
\[ \quad \left( \forall p(p \in is-name-main-part(o)) \rightarrow ; \right. \]
\[ \quad \forall p(p \in is-name-simple-part(o)) \rightarrow ; \]
\[ \quad \forall p(p \in is-name-function(o)) \rightarrow ; \]
∀p(p ∈ is-name-external-function(o)) →
∀p(p ∈ is-name-external-simple-part(o)) →
∀p(p ∈ is-name-reduction-function(o)) →
∀p(p ∈ is-name-standard-function(o)) →
∀p(p ∈ is-input-scalar(o))
∧∃u(u ∈ is-name-scalar(p)) ∧ ∃v(v ∈ is-name-file(p)))
→ LOCAL-IN-NAMES(u, v, o, Files);
∀p(p ∈ is-input-scalar(o))
∧∃u(u ∈ is-name-scalar(p)))
→ LOCAL-IN-NAMES(u, 'norma.dat', o, Files);
∀p(p ∈ is-input-on-domain(o))
∧∀u(u ∈ is-input-on-domain(p)) ∧ ∃v(v ∈ is-name-file(p)))
→ LOCAL-IN-NAMES(u, v, o, Files);
∀p(p ∈ is-input-on-domain(o))
∧∀u(u ∈ is-input-on-domain(p)))
→ LOCAL-IN-NAMES(u, 'norma.dat', o, Files);
∀p(p ∈ is-identifier(o)) → ReplaceAll(p, uname(P), o)

Auxiliary function LOCAL-IN-NAMES(u, v, o, Files) carries out agreed renaming of the names of input variables u localized in part o and input from file v, and the names of input variables set in input files Files.

LOCAL-IN-NAMES(u, v, o, Files) ~
(∃p(p ∈ Files ∧ v = p ∧ ∃q(q ∈ is-name-scalar(p) ∧ q = u))
→ t := UNAME(P); ReplaceAll(q, t, p); ReplaceAll(u, t, o);
∃p(p ∈ Files ∧ v = p ∧ ∃q∃r(q ∈ is-name-variable-on-domain(p) ∧ q = u
∧ r ∈ is-list-range-of-index ∧ q • (• r • )))
→ RENAME-IND(r, p, o)

Auxiliary function RENAME-IND(r, p, o) carries out agreed renaming of input variable’s indices r localized in the part o and input from file p.

RENAME-IND(r, p, o) ~
(∀w(w ∈ is-name-index(r))
→ t := UNAME(P); ReplaceAll(w, t, p); ReplaceAll(w, t, o))

◊ Checking non-recurrence of domains' declarations NONREC-DOMAIN-DECLARATION(o).

Domain with name o1 is said to be depended on domain with name o2 in part o if either the name of domain o2 or the name of domain o3 depended on o2 is included textually into declaration-of-domain o1. Let this character be checked by function DEPENDENCE-DOMAIN(o1, o2, o3).
\[
\text{DEPENDENCE-DOMAIN}(o_1, o_2, o_3) \sim \\
\exists p \exists q \left( p \in \text{is-declaration-of-domain}(o) \wedge \\
q \in \text{is-domain-product}(p) \wedge p = o_1 \cdot \cdot \cdot (\cdot q \cdot \cdot ) \\
\wedge (o_2 \propto q \vee (o_3 \propto q \wedge \text{DEPENDENCE-DOMAIN}(o_3, o_2, o))) \\
\forall q \in \text{is-new-domain}(p) \wedge p = o_1 \cdot q \\
\wedge (o_2 \propto q \vee (o_3 \propto q \wedge \text{DEPENDENCE-DOMAIN}(o_3, o_2, o))) \\
\forall o_1 \in \text{is-name-of-conditional-domain}(p) \\
\wedge (o_2 \in \text{is-name-of-domain}(p) \wedge \text{DEPENDENCE-DOMAIN}(o_3, o_2, o)))
\]

There must be no interdependencies of the domains (recursive declarations of domain).

\[
\text{NONREC-DOMAIN-DECLARATION}(o) \sim \\
\forall o_1 \forall o_2 (o_1, o_2 \in \text{is-name-of-domain}(o) \wedge o_1 \neq o_2 \\
\equiv \neg (\text{DEPENDENCE-DOMAIN}(o_2, o_1, o))
\]

\(\circ\) Checking declaration's being and being unique \(\text{EXIST-UNIQ-DECLARATION}(o)\).

In each part any name (except name-of-index iteration) must be declared.

\[
\text{EXIST-UNIQ-DECLARATION}(o) \sim \\
\forall p \left( p \in \text{is-name-of-domain}(o) \equiv \exists q (q \propto \text{is-declaration-of-domain}(o) \\
\wedge \text{NAME-DOMAIN-DECLARE}(p, q)) \\
\forall p \in \text{is-name-of-scalar}(o) \equiv \\
\exists q \exists r (q \in \text{is-name-of-scalar}(o) \\
\wedge r \in \text{is-declaration-of-scalar-variables}(o) \wedge q \propto r \wedge p = q) \\
\forall p \in \text{is-name-of-variable-on-domain}(o) \equiv \\
\exists q \exists r (q \in \text{is-name-of-variable-on-domain}(o) \\
\wedge r \in \text{is-declaration-of-variables-on-domains}(o) \wedge q \propto r \wedge p = q) \\
\forall p \in \text{is-name-of-index-construction}(o) \equiv \\
\exists q \exists r (q \in \text{is-name-of-index-construction}(o) \\
\wedge r \in \text{is-declaration-of-index-construction}(o) \wedge q \propto r \wedge p = q) \\
\forall p \in \text{is-name-of-domain's-parameters}(o) \equiv \\
\exists q \exists r (q \in \text{is-name-of-domain's-parameters}(o) \\
\wedge r \in \text{is-declaration-of-domain's-parameters}(o) \wedge q \propto r \wedge p = q) \\
\forall p \in \text{is-name-of-external-function}(o) \equiv \\
\exists q \exists r (q \in \text{is-name-of-external-function}(o) \\
\wedge r \in \text{is-declaration-of-external-functions}(o) \wedge q \propto r \wedge p = q) \\
\forall p \in \text{is-name-of-external-simple-part}(o) \equiv
\]
\[ \exists ! q \exists ! r (q \in \text{is-name-of-external-parts}(o) \wedge r \in \text{is-declaration-of-external-simple-parts}(o) \wedge q \equiv r \wedge p = q) \]
\[ \forall p \in \text{is-name-of-index}(o) \equiv \exists q \exists r (q \in \text{is-name-of-index}(o) \wedge r \in \text{is-one-dimensional-domain}(o) \wedge q \equiv r \wedge p = q) \]

Function checking name-of-domain declaring in declaration-of-domain \( q \)

\[ \text{NAME-DOMAIN-DECLARE}(p, q) \sim \exists s (r \in \text{is-s}(q) \wedge q = p \cdot \ldots \cdot r) \]
\[ \forall s, \exists s_1 (r \in \text{is-s}_1(q) \wedge u \in \text{is-s}_2(q) \wedge (q = p \cdot \ldots \cdot u \cdot \ldots \cdot r \vee q = u \cdot \ldots \cdot p \cdot \ldots \cdot r)) \]

\( \square \) Eliminating macroindices \( \text{MACRO-INDEX-EXCLUDE}(o) \).

Any name-of-index-construction included into index expressions is substituted to corresponding, list-explicit-id-expression which is declared in declaration-of-index-construction. Then declaration-of index-construction is eliminated from the text of the program.

\[ \text{MACRO-INDEX-EXCLUDE}(o) \sim (\forall p (p \in \text{is-declaration-of-index-construction}(o) \wedge q \in \text{is-name-of-index-construction}(p) \wedge r \in \text{is-list-explicit-ind-expression}(p)) \rightarrow \text{ReplaceAll}(q,r,o); \text{Delete}(p,o)) \]

\( \square \) Constructing index space \( \text{DEF-INDEX-SPACE}(o) \).

The order of directions (of axes) of index space coincides with the arrangement of the index names in declaration-of-domains’-indices (from the left to the right). If there is no declaration-of-domains’-indices then the order of the directions is defined by the arrangement of names-of-indices in the text of the program.

\[ \text{DEF-IND-SPACE}(o) \sim T \coloneqq \text{GET-PART-NAME}(o); \]
\[ (\forall p (p \in \text{is-declaration-of-domains'-indices}(o) \wedge q \in \text{is-list-name-of-index}(p)) \rightarrow \text{ReplaceAll}(s, q, o); \text{AddListIndex}(q, T); \text{Delete}(p, o); \]
\[ \forall q (q \in \text{is-name-of-index}(o)) \rightarrow \text{AddIndex}(q, T); \]
\[ \text{IndSpace} \coloneqq \text{IndSpace} \cup \{T\}; \]

Function \( \text{GET-PARTNAME}(o) \) give the name of part \( o \)

\[ \text{GET-PARTNAME}(o) \sim (\text{is-main-part}(o) \wedge \exists n (\text{is-name-of-main-part}(n) \wedge \text{MAIN PART} \cdot n \equiv o) \rightarrow n; \]
\[ (\text{is-simple-part}(o) \wedge \exists n (\text{is-name-of-simple-part}(n) \wedge \text{PART} \cdot n \equiv o) \rightarrow n; \]
\[ (\text{is-part-function}(o) \wedge \exists n (\text{is-name-of-part-function}(n) \wedge \text{FUNCTION} \cdot n \equiv o) \rightarrow n. \]
Operator $AddListIndex(q,T)$ puts list of indices $q$ into index space $T$ and if relation $q <_o r$ is true then $q <_r r$ where $r \in \text{is}_-\text{list}_-\text{name}_-\text{-of}_-\text{-index}(o)$.

Operator $AddIndex(q,T)$ puts name of index $q$ into index space $T$ (if it hasn't been there yet) and if relation $q <_o r$ is true then $q <_r r$ where $r \in \text{is}_-\text{name}_-\text{-of}_-\text{-index}(o)$ for $o$.

☐ Substituting domains’ parameters $\text{PAR-DOMAIN-EXCLUDE}(o)$.

The values of domains’ parameters are substituted instead of the names of domains’ parameters in the text of the part.

$\text{PAR-DOMAIN-EXCLUDE}(o) \sim \forall p (p \in \text{is}-\text{declaration}_-\text{-of}_-\text{-domains'}-\text{parameters}(o) \land q \in \text{is}_-\text{name}_-\text{-of}_-\text{-domain'}-\text{s}_-\text{-parameter}(p) \land r \in \text{is}_-\text{integer}_-\text{-without}_-\text{-sign}(p) \land q \bullet = \bullet r \propto p) \rightarrow \text{ReplaceAll}(q,r,o); \text{Delete}(p,o))$

☐ Calculating constant expressions $\text{CONST-EXPR-EVALUATE}(o)$.

The values of all the constant expressions are computed.

$\text{CONST-EXPR-EVALUATE}(o) \sim (\forall p (p \in \text{is}_-\text{constant}_-\text{-expression}(o)) \rightarrow \text{EVALUATE-EXPR}(p) \land \forall p (p \in \text{is}_-\text{constant}_-\text{-expression}_-\text{-without}_-\text{-sign}(o)) \rightarrow \text{EVALUATE-EXPR}(p))$

Function $\text{EVALUATE-EXPR}(e)$ computes the value of expression. Semantics of computing the expression has traditional meaning and may be defined by a well-known algorithm of translating arithmetical expression into Polish notation [12] and the rules of this notation’s interpretation.

☐ Making domains standard $\text{STANDARD-DOMAIN}(o)$.

The main purpose of domains’ standardisation is to reduce all the declarations of rectangular domains to the form.

declaration-of-rectangular-domain:
multidimensional-domain
multidimensional-domain:
one-dimensional-domain
name-of-one-dimensional-domain: (domain-product)
domain-product:
component-domain { ; component-domain }
component-domain:
one-dimensional-domain
(domain-product)
one-dimensional-domain:
name-of-one-dimensional-domain: (name-of-index=value)

$\text{STANDARD-DOMAIN}(o) \sim \text{IMPLICIT-NAMES}(o);
\text{DEF-DOMAIN-EXCLUDE}(o)$;
NAME-DOMAIN-REPLACE\((o)\);
RECT-N-DOMAIN-TRANSFORM\((o)\)

Function **IMPLICIT-NAMES\((o)\)** carries out the following transformation: unnamed *new-domains-without-name* defined in operators **ASSUME, INPUT, OUTPUT** are substituted in these operators to the unique names, declarations of these domains are included into the program under corresponding names.

**IMPLICIT-NAMES\((o)\) ~**
\[
(\forall p ((p \in \text{is-operator} \land \text{ASSUME}(o)) \land (q \in \text{is-declaration} \land \text{-of-} \text{-input}(o)) \land (q \in \text{is-declaration} \land \text{-of-} \text{-output}(o))) \\
\land \exists q (q \in \text{is-new-domain} \land \text{-without-name}(p))) \\
\rightarrow \text{name} := \text{uname}(P); \text{Replace}(q, \text{name}, p); \text{Include}(\text{name} \bullet \bullet q, o))
\]

Function **DEF-DOMAIN-EXCLUDE\((o)\)** carries out the following transformation: declaration \(\text{def}_1\) of every multidimensional-domain with name \text{name-of-multidimensional-domain} and every one-dimensional domain with name \text{name-of-one-dimensional-domain} used as the component of declaration \(\text{def}_2\) of another multidimensional-domain are included as the addition into the text of part \(o\). Name-of-multidimensional-domain or name-of-one-dimensional-domain included into \(\text{def}_1\) is eliminated from \(\text{def}_2\).

**DEF-DOMAIN-EXCLUDE\((o)\) ~**
\[
(\forall p ((p \in \text{is-declaration} \land \text{-of-} \text{-rectangular-domain}(o)) \\
\land (q \in \text{is-name-of-multidimensional-domain}(p)) \land (q \in \text{is-name-of-one-dimensional-domain}(p))) \\
\land \exists s_1 \exists s_2 \exists s_3 (r_1 \in \text{is-s}(p) \land r_2 \in \text{is-s}(p) \land r_3 \in \text{is-s}(p)) \\
\land p = r_1 q \bullet (\bullet r_2 \bullet) \bullet r_3) \\
\rightarrow \text{Replace}(q \bullet ((\bullet r_2 \bullet), (\bullet r_2 \bullet), p); \text{Include}(q \bullet (\bullet r_2 \bullet), o))
\]

Function **NAME-DOMAIN-REPLACE\((o)\)** carries out the following transformation: every name-of-multidimensional-domain and every name of one-dimensional-domain used as the component of declaration \(\text{def}_2\) of another multidimensional-domain is substituted to corresponding to these names declaration of multidimensional-domain or one-dimensional-domain in declaration \(\text{def}_2\).

**NAME-DOMAIN-REPLACE\((o)\) ~**
\[
(\forall p \forall q (p \in \text{is-declaration} \land \text{-of-} \text{-rectangular-domain}(o)) \\
\land (q \in \text{is-name-of-multidimensional-domain}(p)) \land (q \in \text{is-name-of-one-dimensional-domain}(p)) \\
\land (q \bullet \approx p \lor q \bullet \approx p) \\
\land \exists u (\text{NAME-DOMAIN-DECLARE}(q, u) \land u = q \bullet \bullet) \\
\rightarrow \text{Replace}(q, v, p))
\]
Function \text{RECT-N-DOMAIN-TRANSFORM}(o) transforms the declarations of rectangular new domains into equivalent declarations of rectangular \textit{multidimensional} domains.

\text{RECT-N-DOMAIN-TRANSFORM}(o) \sim \\
(\forall p (p \in \overline{\text{is-declaration-of-rectangular-domain}}(o) \land \overline{\text{is-new-domain}}(p) \\
\land \exists u \exists v (u \in \overline{\text{is-name-of-rectangular-domain}}(p) \\
\land v \in \overline{\text{is-list-modification}}(p) \land p = q \bullet \bullet u \bullet / \bullet v) \\
\rightarrow (\overline{\text{is-name-of-one-dimensional-domain}}(u) \land \exists w (\overline{\text{NAME-DOMAIN-DECLARE}}(u, w)) \\
\rightarrow \overline{\text{Replace}}(q \bullet \bullet u \bullet / \bullet v, q \bullet \bullet \overline{\text{MODIFY}}(w, v, o, p)); \\
\overline{\text{is-name-of-multi-dimensional-domain}}(u) \land \exists w (\overline{\text{NAME-DOMAIN-DECLARE}}(u, w)) \\
\rightarrow \overline{\text{Replace}}(q \bullet \bullet u \bullet / \bullet v, q \bullet \bullet \overline{\text{MODIFY}}(w, v, o, p))); \\
\overline{\text{NAME-DOMAIN-REPLACE}}(o))

Function \text{MODIFY}(w, v, o) chooses the next element from the list of modifications \(v\).

\text{MODIFY}(w, v, o) \sim \\
(\overline{\text{is-modification}}(v) \rightarrow \overline{\text{MDFONE}}(w, v, o); \\
\overline{\text{is-list-modification}}(v) \rightarrow \overline{\text{MDFONE}}(w, \text{head}(v), o); \overline{\text{MODIFY}}(w, \text{tail}(v), o))

Functions \text{MDFONE}(w, v, o) and \text{CORRECT}(n, g) correct the domain according to modification \(v\).

\text{MDFONE}(w, v, o) \sim \\
(\exists m \exists n (m \in \overline{\text{is-name-index}}(v) \land n \in \overline{\text{is-value}}(v) \land v = m \bullet = \bullet n \\
\land \exists a (a \in \overline{\text{is-value}}(w) \land w = m \bullet = \bullet a) \\
\rightarrow \overline{\text{Replace}}(m \bullet = \bullet a, m \bullet = \bullet n, w); \\
\exists m \exists g (m \in \overline{\text{is-name-of-one-dimensional-domain}}(v) \land g \neq \emptyset \land v = m \bullet g \\
\land \exists c \exists d (c \in \overline{\text{is-name-index}}(o) \land d \in \overline{\text{is-value}}(o) \land m \bullet \bullet (c \bullet = \bullet d) \\
\rightarrow \overline{\text{MDFONE}}(w, c \bullet = \bullet \overline{\text{CORRECT}}(d, g))))

\text{CORRECT}(n, g) \sim \\
(\exists a \exists b (\overline{\text{is-int-constant}}(a) \land \overline{\text{is-int-constant}}(b) \land n = a \bullet \ldots \bullet b) \\
\rightarrow (\exists c (\overline{\text{is-int-constant}}(c) \land g = + \overline{\text{LEFT}}(c \bullet c) \rightarrow \text{evaluate-expr}(a - c) \bullet \ldots \bullet b) \\
\exists c (\overline{\text{is-int-constant}}(c) \land g = - \overline{\text{LEFT}}(c \bullet c) \rightarrow \text{evaluate-expr}(a + c) \bullet \ldots \bullet b) \\
\exists c (\overline{\text{is-int-constant}}(c) \land g = - \overline{\text{RIGHT}}(c \bullet c) \rightarrow a \bullet \ldots \bullet \text{evaluate-expr}(b - c))}
\exists c \text{ (is-\ int-\ constant(c))} \land g = +\text{RIGHT( } \bullet c \bullet ) \rightarrow a \cdot \ldots \cdot \text{ evaluate-\ expr(b + c) ) )}

Making **ASSUME operators standard** standard-\textbf{ASSUME(o)}.

Every operator \textbf{ASSUME} consists of several relations is transformed into equivalent sequence of operators where each operator \textbf{ASSUME} consists of a relation.

\textbf{standard-ASSUME(o)} ≈
\[
( \forall p \in \text{is-operator - ASSUME}(o) \land q \in \text{is-name-domain}(p) \\
\land \text{FOR } \bullet q \bullet \text{ ASSUME } \propto p ) \\
\rightarrow \text{ReplaceAll( , , FOR } \bullet q \bullet \text{ ASSUME }, p ))
\]
2.6 Definition of translator \( Tr(P_i) \)

Translator \( Tr \) carries out preparation of abstract program \( A(P) \) for \( AM \) machine on standard representation \( P_i \), or fixes semantic errors.

\[
\begin{align*}
Tr(P_i) \sim (\forall o \in \overline{is \cdot part}(P)) \rightarrow \\
\text{RECT-M-DOMAIN-EVALUATE}(o); \\
\text{DIAG-DOMAIN-EVALUATE}(o); \\
\text{VAR-DECLARATION}(o); \\
\text{REBUILD-INDEX}(o); \\
\text{COND-DOMAIN-EVALUATE}(o); \\
\text{INPUT-DECLARATION}(o); \\
\text{OUTPUT-DECLARATION}(o)); \\
\text{INPUT-DATA-PROCESSING}(\text{Files}); \\
\text{CHECK-INPUT-DATA}(P); \\
\text{CALL-GRAPH-CREATE}(P); \\
\text{CHECK-CALL-GRAPH}(P); \\
(\forall o \in \overline{is \cdot part}(P)) \rightarrow \\
\text{OPERATORS-TO-RELATIONS}(o) 
\end{align*}
\]

2.7 Specification of translator's operators

\~ Computing rectangular multidimensional domains \( \text{RECT-M-DOMAIN-EVALUATE}(o) \)

Declarations of rectangular multidimensional domains are transformed into the tuples of relation \( \text{Domains} \). The order of indices in domains' declaration is determined by the indexes' order set in \( \text{Indspace} \). Coordinates of the points from index space making up rectangular multidimensional domain are calculated statically.

Function \( \text{RECT-M-DOMAIN-EVALUATE} \) uses supplementary structures:

\[
\text{DomainNotOrdered}(q) = q \circ (u_1, v_1) \circ \cdots \circ (u_n, v_n), \text{ where} \tag{6}
\]

\( q \cdot \text{name-of-domain} \),

\( u_i, v_i \cdot \text{name-of-index} \) and its value,

\( n \cdot \text{dimension of domain}; \)

\[
\text{DomainOrdered}(q) = q \circ \text{Iname} \circ v_1 \circ \cdots \circ v_n, \text{ where} \tag{7}
\]

\( q \cdot \text{name-of-domain} \),

\( u_i, v_i \cdot \text{name-of-index} \) and its value, \( \text{Iname} = u_1 \circ \cdots \circ u_n \), where if \( u_i < _{\text{Indspace}(o)} \), then

\( u_i < _{\text{DomainOrdered}(q)} u_j; \)

\( n \cdot \text{dimension of domain}. \)

\( \text{RECT-M-DOMAIN-EVALUATE}(o) \sim \)

\[
(\forall p(p \in \overline{is \cdot declaration \cdot of \cdot rectangular \cdot domain}(o) \\
\land is\cdot multidimensional\cdot domain(p) \\
\land q \in \overline{is \cdot name \cdot of \cdot domain}(p)) \\
\rightarrow \text{DomainNotOrdered}(q) := q; \\
(\forall u \forall v(u \in \overline{is \cdot name \cdot of \cdot index}(p) \land v \in \overline{is \cdot value}(p) \land u \bullet = \bullet v \preceq p) \\
\rightarrow \text{DomainNotOrdered}(q) := \text{DomainNotOrdered}(q) \circ (u, v)); 
\]
ORDER-INDEX (DomainNotOrdered(q), IndSpace(o), DomainOrdered(q));
MAKE-DOMAIN (DomainOrdered(q), R);
T ⊗ R ADD-TO Domains;
Delete(p, o))

Function ORDER-INDEX (DomainNotOrdered(q), IndSpace(o), DomainOrdered(q)) transforms representation (6) into (7).
Function MAKE-DOMAIN (DomainOrdered(q), R) forms auxiliary "basic" relation R for domain q on representation (7). This relation differs from Domains by absence of attribute Cond.

MAKE-DOMAIN (DomainOrdered(q), R) ~
R := q ⊗ Iname(q) ⊗ GENERE(v_1) ⊗ ⋯ ⊗ GENERE(v_n)
GENERATE(u * ⋯ * v) ~ F(Π(I), u ≤ I ≤ v)

- Computing diagonal domains DIAG-DOMAIN-EVALUATE(o).
Declarations of diagonal domains are transformed into the tuples of relation Domains. The order of indices in domains' declaration is determined by the order of indices set in IndSpace. Coordinates of the points from index space formed new rectangular domain are computed statically.
Function DIAG-DOMAIN-EVALUATE uses the representation in Domains for unconditional-domain p obtained earlier. Then unconditional domain p is modified into diagonal-domain.

Differed from functions RECT-M-DOMAIN-EVALUATE and RECT-N-DOMAIN-EVALUATE function DIAG-DOMAIN-EVALUATE puts only those values of indices from domain p into relation Domains which satisfy conditions-on-indices set in declaration of diagonal domain.

DIAG-DOMAIN-EVALUATE(o) ~
(∀p(p ∈ is-declaration-of-diagonal-domain(o)
∧ ∃ u∃ v∃ q(is-name-of-unconditional-domain(p)
∧ q ∈ is-name-of-diagonal-domain(p))
∧ v ∈ is-list-condition-on-domain(p)
∧ p = q * u * l * v))
→ R := F(Domains(Cond), D = u);
T ⊗ q ⊗ F(R, ReplaceAll(_, , v)) ADD-TO Domains;
Delete(p, o))

- Declaring the variables VAR-DECLARATION (o).
Declaration of scalar variables and variables on domain are transformed into tuples of relation Variables. For every variables on domain mentioned in the list of iterated variables name of iteration index is indicated as attribute Iter.

VAR-DECLARATION (o) ~
(∀p(p ∈ is-declaration-of-scalar-variables(o)
∧ ∀u(u ∈ is-name-of-scalar(p))))
→ (∀q(q ∈ is-header-of-iteration(o)
∧∃ r (r ∈ is-name-of-iteration-index(q))
∧ u ∋ q → ∅ u ∪ r ADD-TO Variables; Delete(p, o);
→ ∅ u ∪ r ADD-TO Variables; Delete(p, o));

∀ p (p ∈ is-declaration-of-variables-on-domain(p))
∧ ∀ u (u ∈ is-declaration-of-variables-on-domain(p))
∧ ∀ v (v ∈ is-name-of-variable-on-domain(u) ∧ v ∈ is-name-of-domain(u)))
→ (∀ q (q ∈ is-header-of-iteration(q))
∧ ∃ r (r ∈ is-name-of-iteration-index(q))
∧ v ∋ q → w v ∪ r ADD-TO Variables; Delete(p, o);
→ w v ∪ r ADD-TO Variables; Delete(p, o))

◦ Restoring indices REBUILD-INDEX(o)

In using variable on domain the default rule of setting index is right (index expressions coincided with the name of index may be omitted). Function REBUILD-INDEX(o) restores the indices set default in the index expressions of the variables declared on domains based on the information about these domains. The indices indicated as the formal parameters of the parts or function, actual parameters set on domain, elements of the list of iterated variables aren’t to be restored.

REBUILD-INDEX(o) ~
(∀ p (p ∈ is-name-of-variable-on-domain(o))
∧ q ∈ is-list-ind-expression(o)
∀ p • [• q •] ∋ o)
→ ReplaceAll(p • [• q •], p • (• REBUILD(q, GET-INDEX(p)) • ), o);
∀ p (p ∈ is-name-of-variable-on-domain(o))
∧ ∃ q ((q ∈ is-formal-parameter-of-part(o) ∧ p ∋ q)
∨ (q ∈ is-formal parameter-of-function(o) ∧ p ∋ q)
∨ (q ∈ is-initial-parameter(o) ∧ p • ON ∋ q)
∨ (q ∈ is-parameter-result(o) ∧ p • ON ∋ q)
∨ (q ∈ is-iterated-element(o) ∧ p ∋ q))
→ ReplaceAll(p, p • (• ReplaceAll(•, GET-INDEX(p)) • ), o))

Auxiliary function GET-INDEX(y) gives indices of variable y in accordance with the definition domain of this variable.

GET-INDEX(y) ~ R₁ := F(Variables(D, X, Iter), X = y);
R₂ := F(Domains(Cond, D, Iname, I), D = R₁(D));
(R₁(Iter) = ∅ ∅ → R₂(Iname);
R₁(Iter) ≠ ∅ ∅ → R₂(Iname) • R₁(Iter))
Auxiliary function \texttt{REBUILD}(q, Iname) changes the list of index expressions \( q \) to \( \text{Iname} \) substituting index expressions set explicitly from \( q \) to \( \text{Iname} \) instead of corresponding indices (substitution is done by function \texttt{CHANGE}).

\[
\text{REBUILD}(q, \text{Iname}) \sim \text{ReplaceAll}(\circ, \text{Iname}), \quad \forall p \in \text{is - ind - expression}(q) \to \text{CHANGE}(\text{Iname}, p)
\]

\[
\text{CHANGE}(\text{Iname}, p) \sim \\
( \exists u \exists v \exists w (\text{name} \cdot \text{of} \cdot \text{index}(u) \land \text{name} \cdot \text{of} \cdot \text{index}(v) \\
\land \text{is - int - constant}(w) \land p = u \bullet = v \bullet + w \\
\to \text{Replace}(u, v \bullet + w, \text{Iname}); \\
\exists u \exists v \exists w (\text{name} \cdot \text{of} \cdot \text{index}(u) \land \text{name} \cdot \text{of} \cdot \text{index}(v) \\
\land \text{is - int - constant}(w) \land p = u \bullet = v \bullet - w \\
\to \text{Replace}(u, v \bullet - w, \text{Iname}); \\
\exists u \exists v \exists w (\text{name} \cdot \text{of} \cdot \text{index}(u) \land \text{name} \cdot \text{of} \cdot \text{index}(v) \land p = u \bullet = v \\
\to \text{Replace}(u, v, \text{Iname}); \\
\exists u \exists v \exists w (\text{name} \cdot \text{of} \cdot \text{index}(u) \land \text{name} \cdot \text{of} \cdot \text{index}(v) \land p = u \bullet = v \bullet - w \\
\to \text{Replace}(u, v \bullet - w, \text{Iname}); \\
\exists u \exists v \exists w (\text{name} \cdot \text{of} \cdot \text{index}(u) \land \text{name} \cdot \text{of} \cdot \text{domain}(p) \\
\land \text{is - condition - on - domain}(p) \\
\land p = u \bullet = v \bullet \bullet w \bullet I \bullet I) \\
\to R\text{name} = F(\text{Domains}, D = w)(\text{Iname}); \\
Rl = F(\text{Domains}, D = w)(I); \\
R\text{Cond} = F(\text{Domains}, D = w)(\text{Cond}); \\
R\text{Yes} = \text{REPLACE - IND - VAL}(I), R\text{name}, Rl; \\
R\text{No} = \text{REPLACE - IND - VAL}(I), R\text{name}, RL; \\
R\text{Cond} \circ R\text{Yes} \circ u \circ R\text{name} \circ Rl \ ADD \ TO \ \text{Domains}; \\
R\text{Cond} \circ R\text{No} \circ v \circ R\text{name} \circ RL \ ADD \ TO \ \text{Domains};
\]

\hspace{1cm} \square \text{Computing conditional domains} \ COND-\text{DOMAIN-EVALUATE}(o).

Declarations of conditional domains are transformed into the tuples of relation \textit{Domains}. Coordinates of the points from index space forming conditional domain are computed dynamically in the process of AM machine working at the expense of interpretation.
Delete(p,o))

Function REPLACE-IND-VAL(l,RName,RI) constructs the relation, which number of tuples is equal to the number of the tuples in relation RI, and the tuples themselves are the result of substituting the values of indices l instead of their names Iname into l (the order of substitution is determined by the order of the l attribute’s values).

□ Declaring inputs INPUT-DECLARATION(o).

Declaration of input scalars and inputs on domain are transformed into tuples of relation Input and the tuples of relation Relations.

INPUT-DECLARATION(o) ~

n := get-partname(o);

( \( \forall p \in \text{is-input-scalar}(o) \land \exists u(\text{is-name-of-scalar}(u) \land u \in p) \) )

\[ \rightarrow v := \text{GET-IN-FILENAME}(p); \]

\[ u \circ \Lambda \circ v \quad \text{ADD-TO Input}; \]

\[ n \circ \varnothing \circ T \circ u \circ \Lambda \circ \text{INPUT ADD-TO Relations}; \]

\[ \text{Delete}(p,o); \]

\[ \forall p(p \in \text{is-inputs-on-domain}(o)) \]

\[ \lambda \forall u \exists w(u \in \text{is-input-on-domain}(p) \land w \in \text{is-name-of-domain}(p)) \]

\[ \rightarrow v := \text{GET-IN-FILENAME}(p); \]

\[ RName := f(Domains, D = w)(Iname); \]

\[ RI := f(Domains, D = w)(l); \]

\[ var := u \bullet (\bullet \text{ReplaceAll}(o,,RName) \bullet); \]

\[ Var := \text{REPLACE-IND-VAL}(var,RName,RI); \]

\[ Var \circ \Lambda \circ v \quad \text{ADD-TO Input}; \]

\[ n \circ \varnothing \circ \varnothing \circ T \circ T \circ Var \circ \Lambda \circ v \quad \text{ADD-TO Relations}; \]

\[ \text{Delete}(p,o) \]

Auxiliary function GET-IN-FILENAME(p) gives the name of input file of input variable p.

GET-IN-FILENAME(p) ~ ( \( \exists v(v \in \text{is-name-of-file}(p)) \rightarrow v \);

\[ \rightarrow '\text{norma.dat}' \)

Semantics of attributes format for input variables isn’t described in the given specifications it is equal to the corresponding semantics of the FORTRAN language.

□ Declaring outputs OUTPUT-DECLARATION(o).

Declarations of output scalars and outputs on domain are transformed into the tuples of relation Output.

OUTPUT-DECLARATION(o) ~

( \( \forall p \in \text{is-output-scalar}(o) \land \exists u(\text{is-name-of-scalar}(u) \land u \in p) \) )

\[ \rightarrow v := \text{GET-OUT-FILENAME}(p); \]

\[ T \circ u \circ \Lambda \circ v \quad \text{ADD-TO Output}; \text{Delete}(p,o); \]

\[ \forall p(p \in \text{is-outputs-on-domain}(o) \]

\[ \lambda \forall u \exists w(u \in \text{is-outputs-on-domain}(p) \land w \in \text{is-name-of-domain}(p)) \]
\(\forall u \exists \ w(u \in is\text{-}output \cdot on\text{-}domain(p) \land w \in is\text{-}name\text{-}of\text{-}domain(p)))\)

\(\rightarrow v := \text{GET-IN-Filename}(p);\)

\(Rname := \text{F}(\text{Domains}, D = w)(\text{Iname});\)

\(RI := \text{F}(\text{Domains}, D = w)(I);\)

\(var := u \cdot (\cdot \text{ReplaceAll}(\cdot, Rname) \cdot \cdot);\)

\(Var := \text{REPLACE-IND} \cdot \forall (var, Rname, RI);\)

\(RCond \circ Var \otimes \otimes v \Rightarrow \text{ADD-TO Output};\)

\(\text{Delete}(p, o)\)

Auxiliary function \(\text{GET-OUT-Filename}(p)\) gives the name of output file for variable \(p\).

\[\text{GET-OUT-Filename}(p) \sim (\exists v(v \in is\text{-}name\text{-}of\text{-}file(p)) \rightarrow v,\]

\(\rightarrow '\text{display}')\)

Semantics of the attributes for output variables in this specification isn't given as it deals with the rules of representing output variables in the external media and is similar to semantics of corresponding specifications of FORTRAN output format.

□ Processing input data files \(\text{INPUT-DATA-PROCESSING}(Files)\).

The values of input scalars and inputs on domain represented in input files \(Files\) are transformed into the tuples of relations \(InputData\).

\[\text{INPUT-DATA-PROCESSING}(Files) \sim\]

\(\forall p(p \in Files \land q \in is\text{-}name\text{-}of\text{-}file(p) \land \forall r(r \in is\text{-}element\text{-}of\text{-}input(p)))\)

\(\rightarrow (\exists \exists u(v \in is\text{-}name\text{-}of\text{-}scalar(r) \land u \in is\text{-}arithm\text{-}constant(r) \land r = v \cdot = \cdot u)\)

\(\rightarrow v \cdot u \otimes q \Rightarrow \text{ADD-TO InputData};\)

\(\exists \exists u \exists w(v \in is\text{-}name\text{-}of\text{-}variable\text{-}on\text{-}domain(r)\)

\(\land u \in is\text{-}list\text{-}range\text{-}of\text{-}index(r)\)

\(\land w \in is\text{-}data(r) \land r = v \cdot (\cdot u \cdot \cdot) = \cdot w)\)

\(\rightarrow Var := \text{MAKE-INDEX}(v, u); \text{GENEX}(w) \Rightarrow \text{ADD-TO Value};\)

\(Var \circ Value \otimes q \Rightarrow \text{ADD-TO InputData});\)

Auxiliary function \(\text{MAKE-INDEX}(v, u)\) constructs the relation with each tuple equal to the name of input variable \(v\) with index-constants which values are defined by the list of \(u\) indexes' ranges. Let list \(u = u_1, \ldots, u_n\), where \(u_i = \text{name-index}, \cdot = \cdot a_i \cdot \cdot b_i\).

\[\text{MAKE-INDEX}(v, u) \sim\]

\(Temp := \text{GENEX}(a_1 \cdot \cdot b_1) \otimes \otimes \text{GENEX}(a_n \cdot \cdot b_n);\)

\(Temp := \text{ReplaceAll}(\cdot, Temp);\)

\(Temp := v \cdot (\otimes Temp \otimes)\)

Function \(\text{GETNEXT}(w)\) gives the next element from the list of data \(w\).

□ Checking input data \(\text{CHECK-INPUT-DATA}(P)\).
Data must be defined in the input files for every variable introduced during the work. I.e. tuple $\beta$ is said to be in relation $InputData$ for every tuple $\alpha$ in relation $Input$, when the value of $\beta$ differs from the value of $\alpha$ only by the value of attribute $Value$. If this condition is satisfied the value of $Value$ is taken from tuple $\beta$ and put into tuple $\alpha$.

\textbf{CHECK-INPUT-DATA}(P) \sim \\
(\neg \exists(\text{Value}) \land \exists(\text{Value})) \land \neg \exists(\text{value file}) \land \exists(\text{var file}) \\
\rightarrow \forall \alpha (\exists InputData \land \alpha = (\text{var file}) \land \exists(\text{var & file})) \\
\rightarrow \beta = (\text{var file});

$\Box$ Designing the graph of parts and functions' calls $\text{CALL-GRAPH-CREATE}(P)$.

Directed graph of parts and functions' calls $G=(V,E)$ consists of set of nodes $V$ and set of arcs $E$. Set $V = \{v_i\}$ consists of the names of parts and functions' called in program $P$. If there is call $v_i$ from part (function) $v_j$, then arc $(v_i, v_j)$ is included into $E$. Every node $v_i$ contains supplementary information consists of the list of formal parameters of part $v_i$ (it may be empty). If there is no declaration of part but there is its call then the error is fixed in designing the graph and $F$ becomes the result of function $\text{CALL-GRAPH-CREATE}(P)$.

$\Box$ Checking the graph of functions and parts' calls for being correct $\text{CHECK-CALL-GRAPH}(P)$.

The graph of parts and functions' calls is correct if it is acyclic with one node which has no reentrant arcs (this node corresponds to the call of the main part of program $P$ and further will be called entry node).

Graph of parts and functions' calls $G=(V,E)$ is checked for being correct.

$\Box$ Translating the operators into relation Relations $\text{OPERATORS-TO-RELATIONS}(o)$.

Operators $\text{ASSUME}$ are unrolled. Operator specifying some rule of computation $F$ for the variable of the program in all the points $D_i$ from computation domain $D$, $i=1,...,\|D\|$ is represented in the form of $\|D\|$ operators. Each operator specifies the rule of computation $F$ in particular point $D_i$ in computation domain $D$.

If operator $\text{ASSUME}$ is set in construction iteration then it provides with the special mark in the field of attribute $InitVal = InitD \cdot C$ when it is included into relation Relation.

The conditions set in the declaration of conditional computation domain $D$ and the condition of the computation's arguments being defined are taken into consideration in putting in the value of attribute $Cond$. The value of variable $X$ with index $I$ is depended on some arguments, let's name them $Arg$, $Arg = X_1(I_1),...,X_k(I_k)$, $X_i \in T$, $I_i \in N$. Then logical function $Def(Arg)$ is added to condition $Cond$. Function $Def(Arg)$ will be true ($\mathbb{T}$) if all $X_i \in Arg$ have already defined (i.e. their value is the value of attribute $Value \neq \Lambda$) in other case it will be false ($\mathbb{F}$). Function $Def(Arg)$ is a built-in function of $AM$-machine and is performed in the process of its work.

Call to the reduction functions and external functions included into arithmetical expressions are eliminated from these arithmetical expressions and are represented by new scalar operators included into the program.

First all the constructions iteration are processed and after them the operators which are not included into the iterations.

\textbf{OPERATORS-TO-RELATIONS}(o) \sim n := \text{GET-PARTNAME}(o); \\
(\forall q (q \in is\text{-iteration}(o))
\[ \rightarrow \text{ITERATION-TO-RELATIONS}(n,q); \text{Delete}(q,o); \]
\[ \forall p (p \in \overline{\text{is-operator}}(o)) \]
\[ \rightarrow \text{OPERATOR-TO-RELATIONS}(p,n,\emptyset,\emptyset); \text{Delete}(q,o)) \]

Here are auxiliary functions performing analysis of construction iteration \( q \).

Function \( \text{ITERATION-STRUCTURE}(q,a,b,c,d,e,i) \) for iteration \( q \) gives value \( \text{T} \), if \( a \) - header of iteration, \( b \) - boundary values of iteration, \( c \) - initial values of iteration, \( d \) - body of iteration, \( e \) - exit condition of iteration, \( i \) - index of iteration:

\[
\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \sim \\
\exists a (\text{is - head - of - iteration}(a)) \\
\land \exists b (\text{is - boundary - values}(b)) \\
\land \exists c (\text{is - initial - values}(c)) \\
\land \exists d (\text{is - body - of - iteration}(d)) \\
\land \exists e (\text{is - exit - conditions}(e)) \\
\land \exists i (\text{is - name-index - iteration}(i)) \\
\land a \cdot b \cdot c \cdot d \cdot e \cdot \text{END ITERATION} \cdot b = q
\]

Functions \( \text{GET-HEAD-ITERATION}(q) \), \( \text{GET-BOUNDARY}(q) \), \( \text{GET-INITIAL}(q) \), \( \text{GET-BODY}(q) \), \( \text{GET-EXIT}(q) \), \( \text{GET-ITERATION-INDEX}(q) \) for iteration \( q \) give correspondingly header, boundary values, initial values, body, exit condition, index.

\[
\text{GET-HEAD-ITERATION}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow a) \\
\text{GET-BOUNDARY}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow b) \\
\text{GET-INITIAL}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow c) \\
\text{GET-BODY}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow d) \\
\text{GET-EXIT}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow e) \\
\text{GET-ITERATION-INDEX}(q) \sim (\text{ITERATION-STRUCTURE}(q, a, b, c, d, e, i) \rightarrow i)
\]

Here is the function processing construction iteration \( q \).

\[
\text{ITERATION-TO-RELATIONS}(n, q) \sim \\
\text{a := GET-HEAD-ITERATION}(q); \\
\text{b := GET-BOUNDARY}(q); \\
\text{c := GET-INITIAL}(q); \\
\text{d := GET-BODY}(q); \\
\text{e := GET-EXIT}(q); \\
\text{i := GET-ITERATION-INDEX}(q); \\
(\forall p (p \in \overline{\text{is-operator}}(b)) \\
\rightarrow \text{Include}(p,c); \text{Include}(p,d); \text{Delete}(p,q)); \\
(\forall p (p \in \overline{\text{is-element - of - initials}}(c) \\
\land \exists a (\text{is-int-const}(a) \land \text{INITIAL} \cdot i = \cdot a \cdot : \sim p \\
\land \forall r (r \sim \overline{\text{is-operator}}(p))) \\
\rightarrow \text{OPERATOR-TO-RELATIONS}(r,n,i,a)); \\
\text{Delete}(c,q);
\]
\( \forall r(r \propto \text{is-operator}(d)) \) 
\( \rightarrow \text{OPERATOR-TO-RELATIONS}(r, n, i, 1) \);
\( \text{EXIT-TO-RELATIONS}(e, n, i, 1) \)

Function \text{OPERATOR-TO-RELATIONS}(p, n, i, a) puts the tuples into relation \text{Relation}. The form of the tuples is determined by the type of operator \( p \) set in the part with name \( n \) (\( i \) - index of iteration or \( \emptyset \), \( a \) - the value of iteration index or \( \emptyset \)).

\text{OPERATOR-TO-RELATIONS}(p, n, i, a) \sim
\begin{align*}
\text{is-operator-ASSUME}(p) & \rightarrow \text{ADD-ASSUME}(p, n, i, a), \\
\text{is-scalar-operator}(p) & \rightarrow \text{ADD-SCALAR}(p, n, i, a), \\
\text{is-call-of-part}(p) & \rightarrow \text{ADD-COMPUTE}(p, n, i, a)
\end{align*}

Function \text{ADD-ASSUME}(p, n, i, a) converts operator \text{ASSUME} \( p \) into the tuples of relation \text{Relations}. Operator \text{ASSUME} \( p \) is set in the part with name \( n \) (\( i \) - index of iteration or \( \emptyset \), \( a \) - the value of iteration index or \( \emptyset \)).

\text{ADD-ASSUME}(p, n, i, a) \sim
\begin{align*}
d & := \text{GET-DOMAIN-NAME}(p); \\
Rcond & := \mathcal{F}(\text{Domains}, D = d)(\text{Cond}); \\
Riname & := \mathcal{F}(\text{Domains}, D = d)(\text{Iname}); \\
RL & := \mathcal{F}(\text{Domains}, D = d)(l); \\
RU & := \text{REPLACE-IND-VAL}(p, Riname, RL); \\
RUnroll & := \text{EXCLUDE-FUNCTION}(RU, n, i, a); \\
RLeft & := \text{GET-LEFT-PART}(RU); \\
RRight & := \text{GET-RIGHT-PART}(RU); \\
RVar & := \text{GET-VAR}(RLeft); \\
(\exists z(\text{is}\text{-relation}(z) \land z \propto p) & \rightarrow RFunc := RRight), \\
\exists z(\text{is}\text{-call}\text{-of}\text{-part}(z) \land z \propto p) & \rightarrow RFunc := RU); \\
RDep & := \text{Def}(\circ \text{GET-DEPENDENCE}(RRight)\circ); \\
RLambda & := RVar; (\forall c \in RLambda \rightarrow \text{UPDATE}(c, A, RVar)); \\
\text{ADD}_<T_\circ \text{TO}_ \circ \text{RELATIONS}
\end{align*}

Function \text{ADD-COMPUTE}(p, n, i, a) converts operator \text{COMPUTE} \( p \) into the tuples of relation \text{Relations}. Operator \text{COMPUTE} \( p \) is set in the part with name \( n \) (\( i \) - index of iteration or \( \emptyset \), \( a \) - the value of iteration index or \( \emptyset \)).

\text{ADD-COMPUTE}(p, n, i, a) \sim
\begin{align*}
q & := \text{EXCLUDE-FUNCTION}(p, n, i, a); \\
l & := \text{GET-LEFT-PART}(q); \\
r & := \text{GET-RIGHT-PART}(q); \\
RVar & := \text{GET-VAR}(l, \emptyset, \emptyset); \\
RFunc & := q; \\
RDep & := \text{Def}(\circ \text{GET-DEPENDENCE}(r)\circ); \\
RLambda & := RVar; (\forall c \in RLambda \rightarrow \text{UPDATE}(c, A, RVar)); \\
\text{ADD}_<T_\circ \text{TO}_ \circ \text{RELATIONS}
\end{align*}
Function `ADD-SCALAR(p, n, i, a)` converts scalar operator $p$ into the tuples of relation `Relations`. Scalar operator $p$ is set in the part with name $n$ ($i$ - index of iteration or $\emptyset$, $a$ - the value of iteration index or $\emptyset$).

```
ADD-SCALAR(p, n, i, a) ~
q := EXCLUDE-FUNCTION(p, n, i, a);
l := GET-LEFT-PART(q);
r := GET-RIGHT-PART(q);
RDep := Def(\circ GET-DEPENDENCE(r) \circ);
n \circ ia \circ RDep \circ T \circ I \wedge r \quad ADD_TO \quad Relations
```

Function `EXIT-TO-RELATIONS(e, n, i, a)` converts operator of iteration's exit condition $e$ into the tuples of relation `Relations`. Operator of iteration's exit condition $e$ is set in the part with name $n$ ($i$ - index of iteration or $\emptyset$, $a$ - the value of iteration index or $\emptyset$).

```
EXIT-TO-RELATIONS(e, n, i, a) ~
q := EXCLUDE-FUNCTION(e, n, i, a);
RDep := Def(\circ GET-DEPENDENCE(q) \circ);
n \circ ia \circ RDep \circ q \circ \emptyset \circ \emptyset \circ \emptyset \quad ADD_TO \quad Relations
```

Function `GET-DOMAIN-NAME(p)` gives the name to the domain of operator `ASSUME p`.

```
GET-DOMAIN-NAME(p) ~ (\exists n (is - name - of - domain(n) \wedge FOR \bullet n \bullet ASSUME \varnothing \rightarrow n)
```

Function `GET-LEFT-PART(p)` gives the left part of $p$ operator's relation or the list of actual parameters-results placed after key-word `RESULTS` of part's call `COMPUTE` in operator $p$.

```
GET-LEFT-PART(p) ~
(\exists m \exists \exists t (is - name - of - domain(n) \wedge is - name - of - simple - part(m)
\wedge is - list - initial - parameter(s)
\wedge is - list - parameter - result(t)
\wedge FOR \bullet n \bullet ASSUME COMPUTE \bullet m \bullet (\bullet s \bullet RESULTS \bullet t) = p)
\rightarrow t;
\exists m \exists \exists t (is - name - of - simple - part(m)
\wedge is - list - initial - parameter(s)
\wedge is - list - parameter - result(t)
\wedge COMPUTE \bullet m \bullet (\bullet s \bullet RESULTS \bullet t) = p)
\rightarrow t;
\exists n \exists m \exists a (is - name - of - domain(n) \wedge is - arithm - expression(a)
\wedge is - variable - on - domain(m)
\wedge FOR \bullet n \bullet ASSUME \bullet m \bullet = \bullet a = p)
\rightarrow m,
\exists m \exists a (is - arithm - expression(a) \wedge is - name - of - scalar(m) \wedge m \bullet = \bullet a = p)
\rightarrow m)
```
Function \textsc{get-right-part}(p) gives the right part of \emph{p} operator's \emph{relation} or the list of actual initial parameters placed before key-word \textsc{results} of \emph{part's call compute} in operator \emph{p}.

\textsc{get-right-part}(p) \sim

\begin{align*}
& \exists n \exists m \exists t (\text{is-name-of-domain}(n) \land \text{is-name-of-simple-part}(m) \\
& \land \text{is-list-initial-parameter}(s) \\
& \land \text{is-list-parameter-result}(t) \\
& \land \text{compute} \bullet m \bullet (\bullet s \bullet \text{results} \bullet t \bullet) = p) \\
& \rightarrow s, \\
& \exists m \exists s (\text{is-name-of-simple-part}(m) \\
& \land \text{is-list-initial-parameter}(s) \\
& \land \text{is-list-parameter-result}(t) \\
& \land \text{compute} \bullet m \bullet (\bullet s \bullet \text{results} \bullet t \bullet) = p) \\
& \rightarrow s, \\
& \exists m \exists a (\text{is-name-of-domain}(n) \land \text{is-arithm-expression}(a) \\
& \land \text{is-variable-on-domain}(m) \\
& \land \text{for} \bullet n \bullet \text{assume} \bullet m \bullet = a = p) \\
& \rightarrow a; \\
& \exists m \exists a (\text{is-arithm-expression}(a) \land \text{is-name-of-scalar}(m) \land m \bullet = a = p) \\
& \rightarrow a)
\end{align*}

Function \textsc{exclude-function}(u, n, i, a) converts calls to the reduction functions and external functions set in operator \emph{u} from the part with name \emph{n} (\emph{i} - index of iteration, \emph{a} - the value of iteration index).

\textsc{exclude-function}(u, n, i, a) \sim

\begin{align*}
& \forall f (f \in \text{is-call-to-reduction-function}(u)) \\
& \rightarrow t := \text{uname}(P); \text{replace}(f, t, u); \text{add-reduct-func}(t, f, n, i, a); \\
& \forall f (f \in \text{is-call-to-external-function}(u)) \\
& \rightarrow t := \text{uname}(P); \text{replace}(f, t, u); \text{add-extern-func}(t, f, n, i, a))
\end{align*}

\textsc{add-reduct-func}(t, v, n, i, a) \sim

\begin{align*}
& \exists q \exists s \exists a (\text{is-name-reduction-function}(q) \land \text{is-name-of-domain}(s) \\
& \land \text{is-arithm-expression}(a) \land q \bullet (\bullet s \bullet \cdot \cdot) = v) \\
& \rightarrow \text{Rname} := \text{F}(\text{Domains}, D = s)(\text{Iname}); \\
& \text{RI} := \text{F}(\text{Domains}, D = s)(\text{I}); \\
& \text{SumUnroll} := \text{replace-ind-val}(a, \text{Rname}, \text{RI}); \\
& \text{SumUnroll} := \text{exclude-function}(\text{SumUnroll}, n, i, a); \\
& \text{RSumDep} := \text{def} \circ \text{get-dependence}(\text{SumUnroll}) \circ; \\
& \text{SumDep} := \text{rel-in-list}(\text{RSumDep}); \\
& n \circ a \circ \text{SumDep} \circ \text{to} \circ \Lambda \circ v \rightarrow \text{add-to-relations}
\end{align*}

Auxiliary function \textsc{rel-in-list}(\text{RSumDep}) represents the tuples of relation \text{RSumDep} in the form of the list of tuples enumerated by comma.
ADD-EXTERN-FUNC\(t, v, n, i, a\) ~
\[
(\exists q \exists s (is\_name=\_extern\_function(q) \land is\_list\_initial\_parameter(s) \land q \bullet (\bullet s \bullet) = v) \\
\rightarrow ExtDep := Def(\circ GET\_DEPENDENCE(s) \circ); \\
n \circ ia \circ ExtDep \circ T \circ l \circ v \rightarrow ADD\_TO \ Relationships)
\]

Function GET-VAR\(l\), \(l\) - variable on domain with indices-constants or list-parameter-result with indices constants creates the list of variables on domain with indices constants where all the variables set in \(l\) are included.

GET-VAR\(l\) ~
\[
(\text{is\_name=\_of\_scalar}(l) \rightarrow l; \\
\text{is\_variable-on-domain}(l) \rightarrow l; \\
\exists p \exists q (is\_name=\_variable\_on\_domain(p) \land is\_domain\_of\_parameter(q) \land l = p \bullet ON \bullet q) \\
\rightarrow LIST-VAR(p, q); \\
\exists p \exists q (is\_iterated\_variable\_on\_domain(p) \land is\_domain\_of\_parameter(q) \land l = p \bullet ON \bullet q) \\
\rightarrow LIST-VAR(p, q))
\]

Function LIST-VAR\((p, q)\) creates auxiliary relation Temp(Cond, D, Iname, l), which tuples set the domain of parameter q and generate the list of variables with name p and indices from relation Temp.

LIST-VAR\((p, q)\) ~
\[
(\text{is\_name=\_of\_unconditional\_domain}(q) \\
\rightarrow Temp = F(Domains, D = q); Temp(l); \\
(\forall c \in Temp) \rightarrow q \bullet (ReplaceAll(c, c) \bullet)); \\
\exists x \exists y \exists a (is\_name=\_of\_unconditional\_domain(x) \land is\_name=\_of\_index(y) \land is\_int\_constant(t(a) \land q = x \bullet I \bullet y \bullet = \bullet a) \\
\rightarrow Temp = F(Domains, D = q \land Iname = y \land I = a); Temp(l); \\
(\forall c \in Temp) \rightarrow q \bullet (ReplaceAll(c, c) \bullet)); \\
\exists x \exists z (is\_name=\_of\_unconditional\_domain(x) \land is\_list\_index\_expression(z) \land q = x \bullet I (\bullet z \bullet)) \\
\rightarrow (\forall y (y \in is\_name=\_of\_index(z) \land \exists a (is\_int\_constant(t(a) \land y \bullet = \bullet a \circ z))) \\
\rightarrow Temp = F(Domains, D = q \land Iname = y \land I = a); Temp(l); \\
(\forall c \in Temp) \rightarrow q \bullet (ReplaceAll(c, c) \bullet));
\]

Function GET-DEPENDENCE\((l)\), \(l\) - list-initial-parameter or arithmetical expression creates the list of arguments Arg for function of AM-machine Def(Arg) where all the variables set in \(l\) are included. Thus in creation of Def attribute’s value from relation Relations the condition of the definite determination of the computation’s arguments is considered and logical function Def(Arg) is added. Function Def(Arg) will be true (T) if all \(X_i \in Arg\) have already defined (i.e. their value is
the value of attribute Value ≠ Λ) in other case it will be false (F). Function Def(Arg) is a built-in function of AM-machine and is performed in the process of its work.

6 GET-DEPENDENCE(l) ~
(is - arithm - expression(l))
→ (\( \forall r (r \in \bar{s} \text{ - variable - on - domain(l)} \vee r \in \bar{r} \text{ - name - of - scalar(l)}) \rightarrow r);
\exists p \exists q (is - name - variable - on - domain(p) \land \text{is - domain - of - parameter}(q)
\land p \cdot \text{ON} \cdot q \propto l
\rightarrow \text{LIST-VAR}(p, q);
\exists p \exists q (is - iterated - variable - on - domain(p) \land \text{is - domain - of - parameter}(q)
\land p \cdot \text{ON} \cdot q \propto l
\rightarrow \text{LIST-VAR}(p, q))

2.8 The rules of AM-machine work
Relation ARelations(Name, ItVal, Def, Cond, Var, Value, Func), ARelations \subseteq Relations will be named active relation of AM-machine.

Transfer of the tuples from relation Relations to ARelations is carried out by built-in functions of AM-machine
CALL-PART((p(actual-parameters)), p - name of part and
NEXT-ITERATION (n • i), n - name of iteration index, i - value of iteration index.

Function CALL-PART((p(actual-parameters)) invokes the tuples of relation Relations corresponding to computations of part p.

Function NEXT-ITERATION (n • i) invokes the tuples of relation Relations corresponding to the next iteration step i+1 in iterative computation by iteration index n.

Definition of these functions are given below.

Interpretation of abstract program A(P) by abstract machine AM consists of the initialization and stages of interpretation.
1. Initialization.
1) Function CHECK-REASSIGNMENT(Relations) is carried out. It checks the condition of single-assigning the value of variable of program P.
2) Function CALL-PART(main) is done. Main - name of entry node of the calls' graph G(V,E).

2. Stages of interpretation.
1) Set ArgDefined is chosen from relation Relation. ArgDefined is a set of tuples which have all the arguments computed but the values of the variables haven't computed yet. The tuples from ArgDefined which computation depends on the condition Cond and Cond=F are eliminated from ArgDefined and ARelations.

\[
\text{ArgDefined} := F(\text{ARelations}, \text{Def} = T \land \text{Value} = \Lambda);
\text{ArgDefined} := \text{ArgDefined} - F(\text{ArgDefined}, \text{Cond} = F);
\text{ARelations} := \text{ARelations} - F(\text{ARelations}, \text{Cond} = F)
\]

2) Arbitrary subset NonDeterm \subseteq \text{ArgDefined} is chosen. Arbitrariness of such a choice is determined by non-determination of AM-machine and allows to specify not the only process of program P fulfillment but the class of all possible computations.
NonDeterm ⊆ ArgDefined

3) Function $\text{func}$ is computed for every tuple $\langle \text{name} \circ \text{ival} \circ \text{def} \circ \text{cond} \circ \text{var} \circ \text{value} \circ \text{func} \rangle$ from NonDeterm. This function is set in the field of attribute $\text{Func}$ of the same tuple. Besides some variants are possible:

a) $\text{func}$ – call of simple part $\text{name}(\text{actual-parameters})$. The following function of AM-machine is carried out in this case:

CALL-PART($\text{name}(\text{actual-parameters})$)

b) $\text{func}$ – call of external function $\text{name}(\text{actual-parameters})$. The following function of AM-machine is carried out in this case:

CALL-PART($\text{name}(\text{actual-parameters RESULT name})$)

c) $\text{func}$ – arithmetical expression, call of reduction function or operator $\text{INPUT}$.

Arithmetical expression and reduction function are computed (semantics of reduction functions coincides with common mathematical definition of these functions), operator $\text{INPUT}$ is carried out and obtained value $\text{val}$ substitutes value $\text{value} = \Lambda$ of considered tuple. Besides if there is tuple $\langle \text{cond} \circ \text{var} \circ \text{value}_i \circ \text{file} \rangle$ in relation Output then the value $\text{value}_i = \Lambda$ of this tuple is substituted to $\text{val}$.

Formal definition of computation from NonDeterm:

\[
\begin{align*}
\text{COMPUTE}(\text{NonDeterm}); \\
\text{COMPUTE}(\text{ndeterm}) \Rightarrow \\
(\text{ndeterm} = \langle \text{name} \circ \text{ival} \circ \text{def} \circ \text{cond} \circ \text{var} \circ \text{value} \circ \text{func} \rangle \\
\rightarrow (\text{is} \cdot \text{call} \cdot \text{-of} \cdot \text{-part}(\text{func}) \\
\rightarrow \text{CALL-PART}(\text{func}); \text{Delete}(\text{ndeterm}, \text{NonDeterm}); \\
\exists \text{is}(\text{is} \cdot \text{name} \cdot \text{-external} \cdot \text{-function}(f) \\
\land \text{is} \cdot \text{-list} \cdot \text{-initial} \cdot \text{-parameter}(f) \land f \circ (\star / \star) = \text{func} \\
\rightarrow \text{CALL-PART}(f \circ (\star / \star \text{RESULTS} / \star)); \text{Delete}(\text{ndeterm}, \text{NonDeterm});) \\
\text{is} \cdot \text{-arithm} \cdot \text{-expression}(\text{func}) \\
\rightarrow \text{value} := \text{EVALUATE-EXPR}(\text{func}); \text{UPDATE}((\text{Output}, \text{value});) \\
\text{is} \cdot \text{-call} \cdot \text{-to} \cdot \text{-reduction} \cdot \text{-function}(\text{func}) \\
\rightarrow \text{value} := \text{REALIZE}(\text{func}); \text{UPDATE}((\text{Output}, \text{value});) \\
\text{func} = \text{INPUT} \\
\rightarrow \text{value} := \text{INPUT}(\text{Input}, \text{Var}); \text{UPDATE}((\text{Output}, \text{value});)
\end{align*}
\]

Specification of function $\text{UPDATE}((\text{Output}, \text{value})$ and $\text{INPUT}(\text{Input}, \text{var})$ is evident enough and accurate definition of function $\text{REALIZE}(\text{func})$ is determined by realization of AM-machine.

4) Set $\text{OutDefined}$ is chosen from relation Output. $\text{OutDefined}$ is a set of tuples which have all the arguments computed. The tuples from $\text{Out Defined}$ which computation depends on the condition $\text{Cond}$ and $\text{Cond} = \text{F}$ are eliminated from $\text{OutDefined}$.

\[
\text{OutDefined} := F((\text{Output}, \text{Value} \neq \Lambda));
\]
OutDefined := OutDefined - F(Output, Cond = F);

5) Arbitrary subset NonDeterminOut ⊆ OutDefined is chosen. The value value is output into file file for every tuple (cond x i o value o file) from NonDeterminOut.

NonDeterminOut ⊆ OutDefined
OutDefined := OutDefined - NonDeterminOut
OUTPUT(NonDeterminOut)

6) Iterative computations set by the tuples of relation ARelations for the next values of iteration indices (if the iterative computations aren't finished in accordance with the condition of an exit from the iteration) are invoked.

ActivIter := F(ARelations, ItVal ≠ Ø Ø);
ActivIterIndex := IT-INDEX(ActivIter(ItVal));
NEXT-ITERATION(ActivIterIndex)

7) Condition CHECK-STOP of AM-machine stop is checked.
   If all the necessary computations are finished and the values of all the variables which are to be computed have been computed then AM-machine finishes working it NORMALLY and passes over to state stop.
   If there are variables which values are to be computed but new computations aren't possible then AM-machine finishes working it AbNORMALLY and passes over to state error.
   In other case the next stage of iteration is carried out.

CHECK-STOP ~
ActivIter := F(ARelations, ItVal ≠ Ø Ø);
(F(ActivIter, Var = Ø)(Cond) = T ∧ ARelations(Value) ≠ Δ
  → stop;
  F(ActivIter, Var = Ø)(Cond) = T ∧ 3 α (α ∈ ARelations(Value) ∧ α ≠ Δ
  → error;
  → nextAMstep)

CHECKING THE CONDITION OF SINGLE-ASSIGNMENT CHECK-REASSIGNMENT(Relations).
   If there exist even two coincided variables which values are to be computed and the conditions of these variables' computation don't come into the conflict then the condition of single-assignment is broken.

CHECK-REASSIGNMENT(Relations) ~
(∃ α₁ ∃ α₂ (α₁, α₂ ∈ Relations
  ∧ α₁ = name₁ o intval₁ ⨅ def₁ o cond₁ o var₁ o value₁ o func₁,
  ∧ α₂ = name₂ o intval₂ ⨅ def₂ o cond₂ o var₂ o value₂ o func₂,
  ∧ var₁ ∩ var₂ ≠ Ø
  ∧ (cond₁ ∧ ¬(cond₂) = F)
  ∧ intval₁ = intval₂
  → F)
Invoking the tuples while calling the part \textbf{CALL-PART}(c).

Correspondence of actual and formal parameters of c part's call is set and the actual parameters are passed to the tuples of the part. The tuples setting iterative computations are initialized and all the tuples are transferred into relation \textit{ARElations} (with the renaming of the local variables).

\textbf{CALL-PART}(c) \sim
( \exists p \exists q \text{ is - name - of - part}(p) \land \text{is - actual - parameters}(p) \land p \cdot (\bullet q \bullet) = c)

\rightarrow \text{RPart} := \text{F(Relations, Name = p)};
\text{f := GET-FORMAL}(p);
\text{RActual := ACTUAL-FORMAL(RPart,f,p,q)};
\text{RReady := INIT-ITERATION(RAll)};
\text{RENAME-IN-CALL(RReady)} \quad \text{ADD-TO} \quad \text{ARElations})

Function \textbf{GET-FORMAL}(p) analyses the graph of calls G(V,E) and give the formal parameters of part p.

Function \textbf{ACTUAL-FORMAL}(RPart,f,p,q) carries out passing actual parameters f to the tuples of relation Rpart from the part with name p and formal parameters q.

\textbf{ACTUAL-FORMAL}(RPart,f,p,q); \sim
\text{CHECK-FORMAL-ACTUAL}(f,q);
( \exists g \exists h \exists s \text{is - list - name}(g) \land \text{is - list - name}(h)
\land g \cdot \text{RESULTS} \cdot h = f
\land \text{is - list - initial - parameter}(r) \land \text{is - list - parameter - result}(s)
\land r \cdot \text{RESULTS} \cdot s = q

\rightarrow \text{Rpart := MAKE-IN}(p,RPart,g,r); \text{Rpart := MAKE-OUT}(p,RPart,h,s))

Function \textbf{MAKE-IN}(p,RPart,g,r) chooses the next elements from the lists of formal initial parameters g and actual initial parameters r.

\textbf{MAKE-IN}(p,RPart,g,r) \sim
(is-name(g) \rightarrow \text{MAKE-IN-PARAMETER}(p,RPart,g,r));
(is-list-name(g) \rightarrow \text{MAKE-IN-PARAMETER}(p,RPart,head(g),head(r));
\text{MAKE-IN}(p,RPart,tail(g),tail(r)));)

Function \textbf{MAKE-OUT}(p,RPart,g,r) chooses the next elements from the lists of formal parameters results g and actual parameters results r.

\textbf{MAKE-OUT}(p,RPart,g,r) \sim
(is-name(g) \rightarrow \text{MAKE-OUT-PARAMETER}(p,RPart,g,r));
(is-list-name(g) \rightarrow \text{MAKE-OUT-PARAMETER}(p,RPart,head(g),head(r));
\text{MAKE-OUT}(p,RPart,tail(g),tail(r));)

Function \textbf{MAKE-IN-PARAMETER}(p,RPart,g,r) modifies the tuples of relation Rpart from the part with name p while substituting initial formal parameter g to initial actual parameter r.
MAKE-IN-PARAMETER\((p, RPart, g, r)\) \sim

\((\text{is - arithm - expression}(r))
\rightarrow \text{val} = \text{EVALUE - EXP}(r) p \ast T \circ g \circ \text{val} \circ \text{val ADD - TO} RPart; \)
\(\text{is - name - of - external - simple - part}(r) \rightarrow \text{ReplaceAll}(g, r, RPart); \)
\(\text{is - name - of - external - function}(r) \rightarrow \text{ReplaceAll}(g, r, RPart); \)
\(\exists n \exists q (\text{is - name - of - variable - on - domain}(n) \land \text{is - domain - of - parameter}(q)
\land r = n \ast ON \ast q)
\rightarrow (\exists x \exists y (\text{is - name - of - unconditional - domain}(x) \land \text{is - list - ind - expression}(y)
\land q = x \ast l \ast y \ast ))
\rightarrow \text{name}_1 := \text{uname}(P); \text{Replace}(q, \text{name}_1, r); \text{new - domain}(\text{name}_1, x, y);
\text{name}_2 := \text{uname}(P);
\text{Include}(\text{VARIABLE \ast name}_2 \ast \text{DEFINED ON \ast name}_1 \ast \ast p);
\text{VAR-DECLARATION}(p);\)
\text{Include}(\text{FOR \ast name}_1 \ast \text{ASSUME \ast name}_2 \ast n = [\ast y \ast ] \ast \ast p);
\text{Include}(\text{FOR \ast name} \ast \text{ASSUME \ast g = name}_2 \ast \ast p);
\text{REBUILD-INDEX}(p); \text{OPERATORS-TO-RELATIONS}(p);
\text{is - name - of - unconditional - domain}(q)
\rightarrow \text{name}_2 := \text{uname}(P);
\text{Include}(\text{VARIABLE \ast name}_2 \ast \text{DEFINED ON \ast q \ast \ast p});\)
\text{VAR-DECLARATION}(p);\)
\text{Include}(\text{FOR \ast q \ast ASSUME \ast name}_2 \ast n = n \ast \ast p);
\text{Include}(\text{FOR \ast q \ast ASSUME \ast g = name}_2 \ast \ast p);
\text{REBUILD-INDEX}(p); \text{OPERATORS-TO-RELATIONS}(p)); \)

Function NEW-DOMAIN\((n, x, y)\) builds new domain with name \(n\) from the domain with name \(x\) and list of modifications \(y\). This function is an analogous of the superposition of functions RECT-N-DOMAIN-TRANSFORM\((o)\) and RECT-N-DOMAIN-EVALUATE\((o)\) defined above.

Function MAKE-OUT-PARAMETER\((p, RPart, g, r)\) modifies the tuples of relation Rpart from the part with name \(p\) while substituting formal parameter-result \(g\) to actual parameter-result \(r\).

MAKE-OUT-PARAMETER \((p, RPart, g, r)\) \sim

\((\text{is - name - of - scalar}(r)) \rightarrow p \ast T \circ T \circ g \circ \text{val} \circ \text{val ADD - TO} RPart; \)
\(\exists n \exists q (\text{is - name - of - variable - on - domain}(n) \land \text{is - domain - of - parameter}(q)
\land r = n \ast ON \ast q)
\rightarrow (\exists x \exists y (\text{is - name - of - unconditional - domain}(x) \land \text{is - list - ind - expression}(y)
\land q = x \ast l \ast y \ast ))
\rightarrow \text{name}_1 := \text{uname}(P); \text{Replace}(q, \text{name}_1, r); \text{new - domain}(\text{name}_1, x, y);
\text{name}_2 := \text{uname}(P);
\text{Include}(\text{VARIABLE \ast name}_2 \ast \text{DEFINED ON \ast name}_1 \ast \ast p);
\text{VAR-DECLARATION}(p);\)
\text{Include}(\text{FOR \ast name}_1 \ast \text{ASSUME \ast n \ast = name}_2 \ast \ast p);
Include(FOR • name₁ • ASSUME • name₂ • = • g • ••, p);
REBUILD-INDEX(p); OPERATORS-TO-RELATIONS(p);
is - name - of - unconditional - domain(q)
→ name₂ := uname(P);
Include(VARIABLE • name₂ • DEFINED ON • q • ••, p);
VAR-DECLARATION(p);
Include(FOR • q • ASSUME • n • = • name₂ • ••, p);
Include(FOR • q • ASSUME • name₂ • = • g • ••, p);
REBUILD-INDEX(p); OPERATORS-TO-RELATIONS(p))

Invoking tuples at the next iteration step NEXT-ITERATION(n).
If the next iteration step on iteration index n is terminated and the exit condition of iteration hasn't been performed then the tuples are invoked for the next iteration step.

NEXT-ITERATION(n) ~
OneIter := F(ARelations, ItVal = n);
(F(OneIter, Var = Ø)(Cond) = F ∧ F(OneIter, Var ≠ Ø)(Value) ≠ Λ
→ GET-NEXT-ITERATION(n);
→ RESULT-ITERATION(n, max(OneIter(ItVal)))

The tuples specifying the body of iteration are chosen from relation Relations.

GET-NEXT-ITERATION(n) ~
NextIter := F(Relations, ItVal = n • 1);
(∀ c ∈ NextIter ∧ c = (name • itval • def • cond • var • value • func)
→ UPDATE(c, (name • n • i + 1 • def • cond • var • value • func), NextIter);
INI-ITERATION(NextIter) ADD-TO ARelations)

Function INI-ITERATION(R) initializes iterative computations set by the tuples of relation R.

INI-ITERATION(R) ~
(∀ c ∈ R ∧ c = (name • itval • def • cond • var • value • func) ∧ itval = Ø Ø
→;
(∀ c ∈ RAll ∧ c = (name • itval • def • cond • var • value • func) ∧ itval = i • a
→ ReplaceAll(i, a, c))

Function RESULT-ITERATION(n, a) substitutes the name of iteration index n to its maximum value a in all the tuples of relation ARelation which don't set iterative computations. Passing the results of iterative computations is done in this way.
There is auxiliary function IT-INDEX (n • i) ~ n.
2.9 The example of $AM$-machine work.

Consider program $P$ where the principal constructions of the NORMA language are given. Note that this program is demonstrative and computational formulae used in the program don't essentially describe any known numerical method.

**MAIN PART One.**

**BEGIN**

Oji= (Ojj; Oii).
Oj: (Oi:(i=1..n-1); Oj:j=1..n-1).
Ojii/0ii/oi/Right(n-3).

**INDEX** i,j.

**DOMAIN PARAMETERS** $n = 4$.

**VARIABLES** a,Ts,Tn DEFINED ON Oij. **VARIABLES** x,y DEFINED ON Oi.

**VARIABLE** Cond DEFINED ON Oij. **VARIABLE** b DEFINED ON Oj.

OjiYES, Ojinot : Oji/ Ts < 0.5.

**INPUT** a(FILE='data.dat') ON Oij, b(FILE='data.dat') ON Oij.

**INPUT** T ON Oij/i=1,j=2..n-1. T ON Oij/j=1,i=2..n-1.

**OUTPUT** T(FILE='results') ON Oij/i=1..n-1,j=1..n-1.

**OUTPUT** x(FILE='results') ON Oij.

**FOR** Oij ASSUME $x = y + \text{SUM((Oij)a*b)}; y = b[i]=i$.

**FOR** Oij/i=2..n-1,j=2..n-1 **ASSUME** $T = T[i-1] + T[j-1]; Tn = \text{SIN}(T)$.

**FOR** Oij/i=1..n-1,j=1..n-1 **ASSUME** $Tn = \text{SIN}(i+j)$.

**FOR** Oij/i=1..n-1,j=2..n-1 **ASSUME** $Tn = \text{SIN}(i-j)$.

**FOR** Oij ASSUME COMPUTE $\text{Two(Ts ON Oij/j=1 RESULT Tn ON Oij/i=1)}$.

**OUTPUT** Tn(FILE='results') ON Oij, Cond ON OjiYES.

**FOR** OjiYES ASSUME Cond = Ts.

**FOR** Ojinot ASSUME Cond = Ts.

**END PART.**

**PART Two.** Ts RESULT Tn

**BEGIN**

Oji(i=1..M). **DOMAIN PARAMETERS** $M = 3$.

**VARIABLES** Ts,Tn,T DEFINED ON Oi. **VARIABLES** Eps.

**INPUT** Eps. **INPUT** T(FILE='data.dat') ON Oi.

**ITERATION** Tn ON N.

**INITIAL** $N = 0$.

**FOR** Oj ASSUME $Tn = Ts$.

**END INITIAL**

**FOR** Oj ASSUME $Tn = Tn[N-1]/M$.

**EXIT WHEN** ABS($Tn[i=3,N]) < Eps.

**END ITERATION** N.

**END PART.**

Input file data.dat:

\[
\begin{align*}
b(j=1..2) &= 5.0, 7.0; \\
a(i=1..3,j=1..3) &= 2.0, 3.0, -4.0, 3.0, 4.0, -1.0; \\
T(i=1..3) &= 3(-1.1); \\
b(3) &= 9.0;
\end{align*}
\]

Input file norma.dat:

\[
\begin{align*}
\text{Eps} &= 1.0E-6; \\
T(i=2..3,j=1) &= 3.3, 4.4; \\
T(i=1,j=2..3) &= 1.1, 2.2;
\end{align*}
\]

The result of performing function $\text{LOCAL-VARIABLES-RENAMER}(o,o=One,Two$ - the program and input files get the form (we consider function $\text{unam}(P)$ engenders unique names by adding zeroes as the last identifier's symbols):

**MAIN PART One.**
BEGIN
Oij0:  Oij00:  Oij0
. Oij0:  (Oij00:  (i=0..n0-1);  Oij0:  (j=0..n0-1)).
Oij0:  Oij0/0=1..n0-2.  Oij0:  Oij00/Oij0-ROM(Rom0-3).
INDEX i0,j0.

DOMAIN PARAMETERS n0 = 4.

VARIABLE a0i0,j0,T0i0,j0n0, T0i0,j0n0 Defined ON Oij0. VARIABLE x0,y0 DEFINED ON Oij0.
VARIABLE Cond0 DEFINED ON Oij0. VARIABLE b0 DEFINED ON Oij0.
OijYES0,  OijNOT0 :  Oij0/0 Ts0 < 0.5.

INPUT a0i0,j0(FILE='data.data') ON Oij0, b0(FILE='data.data') ON Oij0.

INPUT T0 ON Oij0/n0=1..j0=2..n0-1, T0 ON Oij0/j0=1..i0=2..n0-1.

OUTPUT T0(FILE='results') ON Oij0/n0=1..j0=0..n0-1.

OUTPUT x0(FILE='results') ON Oij0.

FOR Oij0 ASSUME x0 = y0 + SUM((Oij00)a0i0,j0b0); y0 = b0(i0=j0).

FOR Oij0/j0=2..n0-1, i0=2..n0-1 SUMMATE T0 = T0[i0=0..1] + T0[j0=0..1]. T00 = SIN(T0).

FOR Oij0/i0=1..j0=1..n0-1 ASSUME Ts0 = SIN(T00+j0).

FOR Oij0/j0=1..i0=2..n0-1 ASSUME Ts0 = SIN(T00-j0).

FOR Oij0 ASSUME

COMPUTE Ts0 ON Oij0/j0=1..i0=10 RESULT Tn0 ON Oij0/j0=10.

OUTPUT T00(FILE='results') ON Oij0, Cond0 ON OijYES0.

FOR OijYES0 ASSUME Cond0 = Tn0.

FOR OijNOT0 ASSUME Cond0 = Ts0.

END PART.

PART Two.  Ts00 RESULT Tn00

BEGIN

Oij00: (i00 = 0..n00).

DOMAIN PARAMETERS n00 = 3.

VARIABLE T0i00,j00.T000 DEFINED ON Oij00. VARIABLE Eps00.

INPUT Eps00.  INPUT T000(FILE='data.data') ON Oij00.

ITERATION Tn00 ON N00.

INITIAL N00 = 0:

FOR Oij00 ASSUME Tn00 = Ts00.

END INITIAL

FOR Oij00 ASSUME Tn00 = Tn00(n00-1)/M00.

EXIT WHEN ABS(Tn00[00=3.N00]) < Eps00.

END ITERATION N00.

END PART.

Input file data.dat:  Input file norma.dat:

b0(i0=1..2) = 5.0, 7.0;  Eps = 1.0E-6;

a0(i0=1..3,j0=1..3) = 2.0, 3.0, -4.0,
3.0, 4.0, -1.0, 3(5.0);

T00(i0=1..3) = 3(-1.1);

b0(j0=3) = 9.0;

The result of carrying out function DEF-INDEX-SPACE(o), o = One. Two— constructing

IndSpace = {IndSpace_i},  i = 1,...,2;

IndSpace_i = One * i0 * j0

IndSpace_j = Two * i00

and deleting declaration INDEX 10,10 from the text of the program.

The result of performing function PAR-DOMAIN-EXCLUDE(o), o = One, Two:

MAIN PART One.

BEGIN
Oj0: { Ojji; Ojii. 

Ojji, Ojii: { Ojji; j0=1, 4-1 } @jii, j0=1, 4-1. 
Oj00;j0=1, 4-2. Oj00, Ojii0, Ojii0-RI TE (4-3). 
VARIABLE a0, T0, Ts0, Tn0 DEClNED ON Oj00. VARIABLE x0, y0 DEClNED ON Ojii. 
VARIABLE Cond0 DEClNED ON Ojji. VARIABLE b0 DEClNED ON Ojji. 
Ojii0, Ojii00: Ojji0, Ts0 < 0.5. 

INPUT a0 (FILE='data.dat') ON Ojji0, b0 (FILE='data.dat') ON Ojji. 
INPUT T0 ON Oji0/j0=1,j0=2, 4-1, T0 ON Oj0/j0=1, j0=2, 4-1. 
OUTPUT T0 (FILE='results') ON Oji0/j0=4-1, j0=4-1. 
OUTPUT x0 (FILE='results') ON Oj00. 

FOR Oj00 ASSUME x0 = y0 + SUM((Oji0)0*b0); y0 = b0[j0=1]. 
FOR Oji0/j0=2, 4-1, j0=2, 4-1 ASSUME T0 = T0[j0=1]+T0[j0-1]; T0 = SIN(T0). 
FOR Oji0/j0=1, j0=1, 4-1 ASSUME Ts0 = SIN(T0+j0). 
FOR Oji0/j0=1, j0=2, 4-1 ASSUME Ts0 = SIN(T0-j0). 

FOR Ojii00 ASSUME 

COMPUTE Two(Ts0 ON Oj0/j0=10 RESULT Tn0 ON Oj0/j0=10). 
OUTPUT Ts0 (FILE='results') ON Oj00, Cond0 ON Ojii0, Ojii00. 

END PART. 

PART Two. Ts00 RESULT Tn00 
BEGIN 
Oj00; {00=1, 3). 
VARIABLE Ts00, Tn00, T0000 DEFINED ON Oj00. VARIABLE Eps00. 
INPUT Eps00. INPUT T000 (FILE='data.dat') ON Oj00. 
ITERATION Tn00 ON N00. 

INITIAL N00 = 0. 
FOR Oj000 ASSUME Tn00 = Ts00. 
END INITIAL 

FOR Oj000 ASSUME Tn00 = Tn000[N00-1]/3. 
EXIT WHEN ABS(Tn00{00=3,N00}) < Eps00. 
END ITERATION N00. 
END PART. 

The result of performing function CONST-EXPR-EVALUATE(o), o=One, Two (part Two 

isn't changed): 

MAIN PART One. 
BEGIN 
Onew0; {j0=1; j0=2, 3). Onew1; {j0=1; j0=2, 3}. Onew2; {j0=3}. Onew3; {j0=2, 3}. Onew4; {j0=1; j0=1, 3}. Onew5; {j0=1; j0=2, 3}. 
Oji0; {j0=1, 2}. Ojii; {j0=1, 2}. 
Oj0; {j0=1, 3}. Ojii0; {j0=1, 3}. 
Oj00; {j0=1, 2}. Oj000; {j0=1, 2}. 
VARIABLE a0, T0, Ts0, Tn0 DEFINED ON Oj00. VARIABLE x0, y0 DEFINED ON Oj00. 
VARIABLE Cond0 DEFINED ON Oj0, VARIABLE b0 DEFINED ON Oj00. 
Ojii0, Ojii00: Oj00, Ts0 < 0.5. 

INPUT a0 (FILE='data.dat') ON Oj00, b0 (FILE='data.dat') ON Oj00. 
INPUT T0 ON Onew0, T0 ON Onew1. 
OUTPUT T0 (FILE='results') ON Onew2. 
OUTPUT x0 (FILE='results') ON Oj00. 

FOR Oj000 ASSUME x0 = y0 + SUM((Oj00)0*b0); FOR Oj000 ASSUME y0 = b0[j0=10]. 
FOR Onew3 ASSUME T0 = T0[j0=1]+T0[j0-1]. FOR Onew3 ASSUME Ts0 = SIN(T0). 
FOR Onew4 ASSUME Ts0 = SIN(T0+j0). FOR Onew50 ASSUME Ts0 = SIN(T0-j0). 

FOR Ojii000 ASSUME 

COMPUTE Two(Ts0 ON Oj00/j0=10 RESULT Tn0 ON Oj00/j0=10). 
OUTPUT Ts0 (FILE='results') ON Oj00, Cond0 ON Ojii0, Ojii000. 

FOR Ojii0 ASSUME Cond0 = Tn0.
FOR OJ NOT0 ASSUME Con0 = Ts0.
END PART.

The result of performing functions \texttt{RECT-DOMAIN-EVALUATE}($o$), \texttt{DIAG-DOMAIN-EVALUATE}($o$), $o$=One, Two is filling in relation Domains:

<table>
<thead>
<tr>
<th>Domains</th>
<th>Cond</th>
<th>D</th>
<th>Iname</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>Onew0</td>
<td>i0 j0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew0</td>
<td>i0 j0</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew1</td>
<td>i0 j0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew1</td>
<td>i0 j0</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew2</td>
<td>i0 j0</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew3</td>
<td>i0 j0</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew3</td>
<td>i0 j0</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew3</td>
<td>i0 j0</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew3</td>
<td>i0 j0</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew4</td>
<td>i0 j0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew4</td>
<td>i0 j0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew5</td>
<td>i0 j0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Onew5</td>
<td>i0 j0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domains</th>
<th>Cond</th>
<th>D</th>
<th>Iname</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Ojijo</td>
<td>i0 j0</td>
<td>3</td>
</tr>
</tbody>
</table>

Besides declaration of the following domains are deleted from the text of part One:

Onew0:((i0=1):(j0=2..3)). Onew1:((i0=1):(j0=2..3)). Onew2:((i0=3):(j0=3)).
Onew3:((i0=2..3):(j0=2..3)). Onew4:((i0=1):(j0=1..3)). Onew5:((j0=1):(i0=1..3)).
Ojijo:((j0=1..3); i0=1..2).
Ojijo:((i0=1..3); (j0=1..3)).
Ojijo:Ojijo:((i0=1..3). Ojijo:j0=1..3).
Ojijo:(j0=1..2). Ojijo:(i0=1..2).

and the declaration of the domain given below is deleted from the text of part Two:

Ojijo:((i0=1..3).

The result of performing functions \texttt{VAR-DECLARATION}($o$) and \texttt{REBUILD-INDEX}($o$), $o$=One, Two - modification of program and filling in relation \texttt{variables}:

**MAIN PART One.**

BEGIN
Ojijo\texttt{YES0 } Ojino\texttt{NOT0 : O}jijo/ Ts0(i0,j0) < 0.5.

INPUT a0(i0,j0)(FILE='data.dat') ON Ojijo, b0(i0)(FILE='data.dat') ON Ojijo.
INPUT T0(i0,j0) ON Onew0. TO(i0,j0) ON Onew1.
OUTPUT TO(i0,j0)(FILE='results') ON Onew2.
OUTPUT x0(i0)(FILE='results') ON Ojijo.

FOR Ojio ASSUME x0(i0) = y0(i0) + SUM(Ojijo a0(i0,j0)*b0(j0)).
FOR Ojio ASSUME y0(i0) = b0(i0).
FOR Onew3 ASSUME TO(i0,j0) = TO(i0-1,j0)+T(i0,j0-1).
FOR Onew3 ASSUME Ts0(i0,j0) = SIN(TO(i0,j0)).
FOR Onew4 ASSUME Ts0(10,j0) = SIN(10+j0).
FOR Onew5 ASSUME Ts0(10,j0) = SIN(10-j0).
FOR Oi0 ASSUME
    COMPUTE Two(Ts0 ON Oi0/j0=10 RESULT Tn0 ON Oi0/10=j0).
    OUTPUT Tn0(10,j0)(FILE='results') ON Oi0, Cond0(10,j0) ON Oi0YES0.
FOR OiY0YES0 ASSUME Cond0(10,j0) = Tn0(10,j0).
FOR OiY0NOT0 ASSUME Cond0(10,j0) = Ts0(10,j0).
END PART.
PART Two. Ts00 RESULT Tn00
BEGIN
INPUT Eps00. INPUT T00(100)(FILE='data.dat') ON Oi00.
ITERATION Tn00 ON N00.
    INITIAL N00 = 0:
    FOR Oi00 ASSUME Tn00(i00,N00) = Ts000(i00).
    END INITIAL
FOR Oi00 ASSUME Tn00(i00,N00) = Tn00[i00,N00-1]/3.
EXIT WHEN ABS(Tn00[3,N00]) < Eps00.
END ITERATION N00.
END PART.

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi0</td>
</tr>
<tr>
<td>Oi00</td>
</tr>
<tr>
<td>Oi00</td>
</tr>
<tr>
<td>Oi00</td>
</tr>
<tr>
<td>$\phi$</td>
</tr>
</tbody>
</table>

The result of performing function $\text{COND-DOMAIN-EVALUATE}(o), o=\text{One}, \text{Two}$ is adding tuples to relation $\text{Domains}$:

<table>
<thead>
<tr>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>$(Ts0(1,1)&lt;0.5)$</td>
</tr>
<tr>
<td>$(Ts0(1,2)&lt;0.5)$</td>
</tr>
<tr>
<td>$(Ts0(2,1)&lt;0.5)$</td>
</tr>
<tr>
<td>$(Ts0(2,2)&lt;0.5)$</td>
</tr>
<tr>
<td>$-\text{(Ts0(1,1)&lt;0.5)}$</td>
</tr>
<tr>
<td>$-\text{(Ts0(1,2)&lt;0.5)}$</td>
</tr>
<tr>
<td>$-\text{(Ts0(2,1)&lt;0.5)}$</td>
</tr>
<tr>
<td>$-\text{(Ts0(2,2)&lt;0.5)}$</td>
</tr>
</tbody>
</table>

and deleting declaration of the following domain from the text of part $\text{One}$
$Oi0YES0 \ , \ Oi0NOT0 : \text{Oi0/Ts0}(10,j0) < 0.5.$
The result of performing function INPUT-DECLARATION(\(o\)), OUTPUT-DECLARATION(\(o\)), \(o=\text{One}, \text{Two}\) - modification of the program and filling in relations Input, Output, Relations.

**MAIN PART One.**
BEGIN
FOR O10 ASSUME \(x_0(10) = y_0(10) + \text{SUM}((O|j|a_0(10, j, 0) * b_0(j, 0))\).  
FOR O10 ASSUME \(ya(0) = b_0(10)\).  
FOR Onew3 ASSUME \(T_0(10, j, 0) = T_0(10-1, j, 0) + T(10, j, 0-1)\).  
FOR Onew3 ASSUME \(T_0(10, j, 0) = \text{SIN}(T_0(10, j, 0))\).  
FOR Onew4 ASSUME \(T_0(10, j, 0) = \text{SIN}(10+j, 0)\).  
FOR Onew5 ASSUME \(T_0(10, j, 0) = \text{SIN}(10-j, 0)\).  
FOR O10 ASSUME  
\[\text{COMPUTE Two}(T_0 \text{ ON } O1j0/j0=10 \text{ RESULT } Tn0 \text{ ON } O1j0/j0=10)\].  
FOR O1j0YES0 ASSUME \(\text{Cond}_0(10, j, 0) = Tn0(10, j, 0)\).  
FOR O1j0NO0 ASSUME \(\text{Cond}_0(10, j, 0) = T_0(10, j, 0)\).
END PART.

**PART Two.** \(T_000 \text{ RESULT } Tn000\)
BEGIN
ITERATION Tn000 ON N00.
INITIAL N00 = 0:
FOR O100 ASSUME \(Tn00(100, N00) = T_00(100)\).
END INITIAL
FOR O100 ASSUME \(Tn00(100, N00) = Tn00[100, N00-1] / 3\).
EXIT WHEN \(\text{ABS}(Tn00[3, N00]) < \text{Eps}00\).
END ITERATION N00.
END PART.

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Value</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0(1,1)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(1,2)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(1,3)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(2,1)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(2,2)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(2,3)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(3,1)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(3,2)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>a0(3,3)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>b0(1)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>b0(2)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>b0(3)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>T0(1,2)</td>
<td>(\lambda)</td>
<td>'norma.dat'</td>
</tr>
<tr>
<td>T0(1,3)</td>
<td>(\lambda)</td>
<td>'norma.dat'</td>
</tr>
<tr>
<td>T0(2,1)</td>
<td>(\lambda)</td>
<td>'norma.dat'</td>
</tr>
<tr>
<td>T0(3,1)</td>
<td>(\lambda)</td>
<td>'norma.dat'</td>
</tr>
<tr>
<td>Eps00</td>
<td>(\lambda)</td>
<td>'norma.dat'</td>
</tr>
<tr>
<td>T00(1)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>T00(2)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td>T00(3)</td>
<td>(\lambda)</td>
<td>'data.dat'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Cond</td>
<td>Var</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>T0(3,3)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>x0(1)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>x0(2)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>x0(3)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(1,1)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(1,2)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(1,3)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(2,1)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(2,2)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(2,3)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(3,1)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(3,2)</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>Tn0(3,3)</td>
</tr>
<tr>
<td></td>
<td>(T0(1.1)&lt;0.5)</td>
<td>Cond0(1.1)</td>
</tr>
<tr>
<td></td>
<td>(T0(1.2)&lt;0.5)</td>
<td>Cond0(1.2)</td>
</tr>
<tr>
<td></td>
<td>(T0(2.1)&lt;0.5)</td>
<td>Cond0(2.1)</td>
</tr>
<tr>
<td></td>
<td>(T0(2.2)&lt;0.5)</td>
<td>Cond0(2.2)</td>
</tr>
<tr>
<td></td>
<td>-(T0(1.1)&lt;0.5)</td>
<td>Cond0(1.1)</td>
</tr>
<tr>
<td></td>
<td>-(T0(1.2)&lt;0.5)</td>
<td>Cond0(1.2)</td>
</tr>
<tr>
<td></td>
<td>-(T0(2.1)&lt;0.5)</td>
<td>Cond0(2.1)</td>
</tr>
<tr>
<td></td>
<td>-(T0(2.2)&lt;0.5)</td>
<td>Cond0(2.2)</td>
</tr>
</tbody>
</table>
## Relations

<table>
<thead>
<tr>
<th>Name</th>
<th>ItVal</th>
<th>Def</th>
<th>Cond</th>
<th>Var</th>
<th>Value</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(1,1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(1,2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(1,3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(2,1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(2,2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(2,3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(3,1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(3,2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>a0(3,3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>b0(1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>b0(2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>b0(3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T0(1,2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T0(1,3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T0(2,1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T0(3,1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>Eps00</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T00(1)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T00(2)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>Θ</td>
<td>T</td>
<td>T</td>
<td>T00(3)</td>
<td>Λ</td>
<td>INPUT</td>
</tr>
</tbody>
</table>

The result of performing function `INPUT-DATA-PROCESSING (Files)` is filling in relation `InputData`.

<table>
<thead>
<tr>
<th>Input Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Var</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>a0(1,1)</td>
<td>2.0</td>
</tr>
<tr>
<td>a0(1,2)</td>
<td>3.0</td>
</tr>
<tr>
<td>a0(1,3)</td>
<td>-4.0</td>
</tr>
<tr>
<td>a0(2,1)</td>
<td>3.0</td>
</tr>
<tr>
<td>a0(2,2)</td>
<td>4.0</td>
</tr>
<tr>
<td>a0(2,3)</td>
<td>-1.0</td>
</tr>
<tr>
<td>a0(3,1)</td>
<td>5.0</td>
</tr>
<tr>
<td>a0(3,2)</td>
<td>5.0</td>
</tr>
<tr>
<td>a0(3,3)</td>
<td>5.0</td>
</tr>
<tr>
<td>b0(1)</td>
<td>5.0</td>
</tr>
</tbody>
</table>

The result of performing function `CHECK-INPUT-DATA(P)` - filling in the values of attribute `Value` in relation `Input`.
The result of performing function \texttt{CALL-GRAPH-CREATE}(P) is designing the graph of functions and parts’ calls $G=(V,E)$, $V = \{\text{One,Two}(T00\text{RESULTS}000)\}$, $V = \{\text{One,Two}\}$. The result of performing function \texttt{CHECK-CALL-GRAPH}(P) - $T$.

The result of performing function \texttt{OPERATORS-TO-RELATIONS}(o) - addition to relation \textit{Relations} (ui-unique names obtained in the result of using function \textit{uname}(P)):
<table>
<thead>
<tr>
<th>Name</th>
<th>ItVal</th>
<th>Def</th>
<th>Cond</th>
<th>Var</th>
<th>Value</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(1,1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(1,2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(1,3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(2,1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(2,2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(2,3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(3,1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(3,2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>a0(3,3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>b0(1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>b0(2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>b0(3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T0(1,2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T0(1,3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T0(2,1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T0(3,1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>Eps00</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T00(1)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T00(2)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊗ ⊗</td>
<td>T</td>
<td>T</td>
<td>T00(3)</td>
<td>⊗</td>
<td>INPUT</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>x0(1)</td>
<td>⊗</td>
<td>y0(1)+u1</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>u1</td>
<td>⊗</td>
<td>SUM((o)0)a0(1,j0)*b0(j0)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>x0(2)</td>
<td>⊗</td>
<td>y0(2)+u2</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>u2</td>
<td>⊗</td>
<td>SUM((o)0)a0(2,j0)*b0(j0)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>x0(3)</td>
<td>⊗</td>
<td>y0(3)+u3</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>u3</td>
<td>⊗</td>
<td>SUM((o)0)a0(3,j0)*b0(j0)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>y0(1)</td>
<td>⊗</td>
<td>b0(1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>y0(2)</td>
<td>⊗</td>
<td>b0(2)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>y0(3)</td>
<td>⊗</td>
<td>b0(3)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,2)</td>
<td>⊗</td>
<td>T0(2,1)+T0(2,1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,3)</td>
<td>⊗</td>
<td>T0(2,1)+T0(2,2)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(2,2)</td>
<td>⊗</td>
<td>T0(2,2)+T0(3,1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(3,1)</td>
<td>⊗</td>
<td>T0(2,3)+T0(3,2)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(3,2)</td>
<td>⊗</td>
<td>SIN(T0(2,2))</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(2,3)</td>
<td>⊗</td>
<td>SIN(T0(2,3))</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(3,2)</td>
<td>⊗</td>
<td>SIN(T0(3,2))</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(3,3)</td>
<td>⊗</td>
<td>SIN(T0(3,3))</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,1)</td>
<td>⊗</td>
<td>SIN(1+1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,2)</td>
<td>⊗</td>
<td>SIN(1+2)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,3)</td>
<td>⊗</td>
<td>SIN(1+3)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(2,1)</td>
<td>⊗</td>
<td>SIN(2-1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(3,1)</td>
<td>⊗</td>
<td>SIN(3-1)</td>
</tr>
<tr>
<td>One</td>
<td>⊗ ⊗</td>
<td>⊗</td>
<td>⊗</td>
<td>T0(1,1), T0(1,2), T0(1,3)</td>
<td>⊗</td>
<td>Two(Ts0 ON Oij0/j0=1 REZULT Tn0 ON Oij0/j0=1)</td>
</tr>
</tbody>
</table>
Design of abstract program $A(P)$ is finally finished. Now here is a brief description of AM-machine working by this program assuming that $NonDeterm=ArgDefined$ and $NonDetermOut=OutDefined$. Thus to eliminate non-determination of AM-machine and not to specify all the variants of possible computations we have to demonstrate semantics of natural/ideal parallelism.

Initialization.

Checking CHECK-REASSIGNMENT(Relations) gives $T$ after that function CALL-PART(One) is performed invoking tuples $F(Relations, Name = One)$.

Stage of interpretation 1.

a) Relation $NonDeterm=ArgDefined$ determines computation variables $a0(1,1), a0(1,2), a0(1,3), a0(2,1), a0(2,2), a0(2,3), a0(3,1), a0(3,2), a0(3,3), b0(1), b0(2), b0(3), T0(1,2), T0(1,3), T0(2,1), T0(2,3), T0(3,1), T0(3,2), T0(3,3), T0(2,1), T0(3,2), T0(3,3)$. The values of these variables are computed corresponding to function $func$ set in the field of attribute $Func$.

b) CHECK-STOP = nextAMstep

Stage of interpretation 2.

a) Relation $ArgDefined$ determines the computation of variables $u1, u2, u3, y0(1), y0(2), y0(3), T0(1,2), T0(1,3), T0(2,1), T0(2,3), T0(3,1), T0(3,2), T0(3,3))$. The values of these variables are computed. Computation of $\{T0(1,1), T0(1,2), T0(1,3)\}$ engenders call of function CALL-PART(Two(Ts0 ON Oij0/0=1 RESULT Tn0 ON Oij0/0=1)) and invoking the tuples of part Two.

An extract of relation Relations is -obtained as the result of invoking the tuples of part Two.
### ARelations

<table>
<thead>
<tr>
<th>Name</th>
<th>WVal</th>
<th>Def</th>
<th>Cond</th>
<th>Var</th>
<th>Value</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>T</td>
<td>T</td>
<td>Eps00</td>
<td>⊥</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>T</td>
<td>T</td>
<td>T00(1)</td>
<td>⊥</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>T</td>
<td>T</td>
<td>T00(2)</td>
<td>⊥</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>T</td>
<td>T</td>
<td>T00(3)</td>
<td>⊥</td>
<td>INPUT</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(Ts0(1,1))</td>
<td>T</td>
<td>u4</td>
<td>⊥</td>
<td>Ts0(1,1)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(Ts0(2,1))</td>
<td>T</td>
<td>u5</td>
<td>⊥</td>
<td>Ts0(2,1)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(Ts0(3,1))</td>
<td>T</td>
<td>u6</td>
<td>⊥</td>
<td>Ts0(3,1)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u4)</td>
<td>T</td>
<td>Ts00(1)</td>
<td>⊥</td>
<td>u4</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u5)</td>
<td>T</td>
<td>Ts00(2)</td>
<td>⊥</td>
<td>u5</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u6)</td>
<td>T</td>
<td>Ts00(3)</td>
<td>⊥</td>
<td>u6</td>
</tr>
<tr>
<td>Two</td>
<td>N00 0</td>
<td>Def(Ts00(1))</td>
<td>T</td>
<td>Tn00(1,0)</td>
<td>⊥</td>
<td>Ts00(1)</td>
</tr>
<tr>
<td>Two</td>
<td>N00 0</td>
<td>Def(Ts00(2))</td>
<td>T</td>
<td>Tn00(2,0)</td>
<td>⊥</td>
<td>Ts00(2)</td>
</tr>
<tr>
<td>Two</td>
<td>N00 0</td>
<td>Def(Ts00(3))</td>
<td>T</td>
<td>Tn00(3,0)</td>
<td>⊥</td>
<td>Ts00(3)</td>
</tr>
<tr>
<td>Two</td>
<td>N00 1</td>
<td>Def(Tn00(1,N00-1))</td>
<td>T</td>
<td>Tn00(1,1)</td>
<td>⊥</td>
<td>Tn00(1,0)/3</td>
</tr>
<tr>
<td>Two</td>
<td>N00 1</td>
<td>Def(Tn00(2,N00-1))</td>
<td>T</td>
<td>Tn00(2,1)</td>
<td>⊥</td>
<td>Tn00(2,0)/3</td>
</tr>
<tr>
<td>Two</td>
<td>N00 1</td>
<td>Def(Tn00(3,N00-1))</td>
<td>T</td>
<td>Tn00(3,1)</td>
<td>⊥</td>
<td>Tn00(3,0)/3</td>
</tr>
<tr>
<td>Two</td>
<td>N00 1</td>
<td>Def(Tn00(3,1,1), Eps00) &amp; ABS(Tn00(3,1))</td>
<td>T</td>
<td>u7</td>
<td>⊥</td>
<td>Tn00(1,N00)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(Tn00(1,N00))</td>
<td>T</td>
<td>u8</td>
<td>⊥</td>
<td>Tn00(2,N00)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(Tn00(2,N00))</td>
<td>T</td>
<td>u9</td>
<td>⊥</td>
<td>Tn00(3,N00)</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u7)</td>
<td>T</td>
<td>Tn00(1,1)</td>
<td>⊥</td>
<td>u7</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u8)</td>
<td>T</td>
<td>Tn00(1,2)</td>
<td>⊥</td>
<td>u8</td>
</tr>
<tr>
<td>Two</td>
<td>⊥ ⊥</td>
<td>Def(u9)</td>
<td>T</td>
<td>Tn00(1,3)</td>
<td>⊥</td>
<td>u9</td>
</tr>
</tbody>
</table>

b) **CHECK-STOP = nextAMstep**

Stage of interpretation 3.

a) relation *ArgDefined* determines the computation of variables x0(1), x0(2), x0(3), T0(2,3), T0(3,2), Ts0(2,2), u4, u5, u6, Eps00, T00(1), T00(2), T00(3). The values of these variables are computed.

b) Relation *OutDefined* determines variables x0(1), x0(2), x0(3), which values has been computed. Output of these values is carried out into file 'results'.

c) **CHECK-STOP = nextAMstep**

Stage of interpretation 4.

a) Relation *ArgDefined* determines the computation of variables T0(3,3), Ts0(2,3), Ts0(3,2) u4, u5, u6, T00(1), T00(2), T00(3). The values of these variables are computed.

b) Relation *OutDefined* determines variable T0(3,3) which value has been computed. Output of these values is carried out into file 'results'.

c) **CHECK-STOP = nextAMstep**

Stage of interpretation 5.

a) Relation *ArgDefined* determines the computation of Ts0(3,3), (Tn0(2,1), Tn0(2,2), Tn0(2,3)), Tn00(1,0), Tn00(2,0), Tn00(3,0). The values of these variables are computed. Computation of (Tn0(1,1), Tn0(1,2), Tn0(1,3)) engenders call of function

CALL-PART(TWO(Ts0 ON Oij0/0=2 RESULT Tn0 ON Oij0/0=2))

and invoking the tuples of part Two (this process and the computations determined by it are not described to make the text shorter).

c) **CHECK-STOP = nextAMstep**

Stage of interpretation 6.

a) Relation *ArgDefined* determines the computation Tn00(1,1), Tn00(2,1), Tn00(3,1), (Tn0(3,1), Tn0(3,2), Tn0(3,3)). The values of these variables are computed. Computation of (Tn0(3,1), Tn0(3,2), Tn0(3,3)) engenders call of function
CALL-PART(Two(Ts0 ON Oij0/0=3 RESULT Tn0 ON Oij0/0=3))
and invoking the tuples of part Two (this process and the computations determined by it aren't
described to make the text shorter).
Exit condition of iteration ABS(Tn00(3,1))<Eps00 is checked and either the next step to
compute Tn00(1,1), Tn00(2,1), Tn00(3,1) is done or passing the result is performed. Let's assume
that iteration is finished.
c) CHECK-STOP = nextAMstep
Stage of interpretation 7.
a) Relation ArgDefined determines the computation of u7, u8, u9. The values of these
variables are computed.
c) CHECK-STOP = nextAMstep
Stage of interpretation 8.
a) Relation ArgDefined determines the computation of Tn0(1,1), Tn0(1,2), Tn0(1,3). The
values of these variables are computed.
b) Relation OutDefined determines variables Tn0(1,1), Tn0(1,2), Tn0(1,3) which values have
been computed. Output of these values is carried out into file 'results'.
c) CHECK-STOP = nextAMstep
Stage of interpretation 9.
a) Relation ArgDefined determines the computation of
Cond0(1,1) (if Ts0(1,1)<0.5),
Cond0(1,2) (ifTs0(1,2)<0.5),
Cond0(1,1) (if ¬(Ts0(1,1)<0.5),
Cond0(1,2) (if ¬(Ts0(1,2)<0.5))
The tuples determining the computation of Cond0(1,1) (if ¬(Ts0(1,1)<0.5)),
Cond0(1,2) (if ¬(Ts0(1,2)<0.5)) are eliminated from
ArgDefined and ARElations as the condition Cond=F.
The values of the variables Cond0(1,1) (if Ts0(1,1)<0.5),
Cond0(1,2) (ifTs0(1,2)<0.5) are computed.

b) Relation OutDefined defines variables Cond0(1,1), Cond0(1,2) which values have been
computed. The tuples describing output Cond0(1,1) (if ¬(Ts0(1,1)<0.5)), Cond0(1,2)
(if ¬(Ts0(1,2)<0.5)) are eliminated from OutDefined as condition Cond=F. Output of the values
Cond0(1,1) (if Ts0(1,1)<0.5), Cond0(1,2) (ifTs0(1,2)<0.5) is carried out into file "results".
c) CHECK-STOP = nextAMstep
Stage of interpretation ...
CHECK-STOP = nextAMstep

2.10 Definite work of AM-machine

The value of variable $X \in ARElations(Var)$ are computable if it has no information
dependencies or the values of all variables $Y_1,...,Y_n$, from which $X_i$ has information dependencies
have been computed.

\[
\text{COMPUTABLE}(X) \land \\
\exists \alpha (\alpha \in ARElations \land \alpha = \langle \text{name} \circ \text{ival} \circ \text{def} \circ \text{cond} \circ \text{var} \circ \text{value} \circ \text{func} \rangle \\
\land X = \text{var} \land \text{LOG-EXPR}(\text{def}) = \text{T} \land \text{LOG-EXPR}(\text{cond}) = \text{T} \land \text{value} \neq \lambda)
\]

(In this definition function LOG-EXPR(\(e\)) computes the value of logical expression \(e\)).

Computation of function func = F(ARelations, Var = X)(Func) if \(\text{COMPUTE}(X) \equiv \text{T}\)
is called computation of the value of variable $X \in ARElations(Var)$.

\[
\text{COMPUTATION}(X) \land \exists \alpha (\alpha \in ARElations \land X \equiv \alpha \land \text{COMPUTE}(X)) \\
\rightarrow \text{COMPUTE}(\alpha))
\]
Variable $X$ is determined definitely if program $P$ satisfies the condition of single-assignment and $X \in A\text{Relations}(\text{Var})$.

\[\text{DEF-ONE-TO-ONE}(X) \implies \text{CHECK-REASSIGNMENT}(\text{Relations})\]

Program $P$ is correct if it is semantically correct and the work of $AM$-machine is terminated.

**Theorem.** All the values of the variables are determined definitely for correct program $P$.

**Proof.**

Work of $AM$-machine finishes in state $\text{stop}$ for correct program so the number of interpretation stages is finite and according to definition of state $\text{stop}$ the values of all the variables which are to be computed are computed.

The condition of single-assignment is fulfilled for correct program as function $\text{CHECK-REASSIGNMENT}$ is done by $AM$-machine. Thus all the variables of program $P$ are determined definitely in accordance with the definition of definite determination.

The arbitrariness of choosing set $\text{NonDeterm} \subseteq \text{ArgDefined}$ at each stage of interpretation and computation of the variables' values from $\text{Nondeterm}$ don't break the definite determination of the variables.

### 2.11 Notes to semantics' specification

1) The specification of the NORMA language's semantics given above is a shortened specification. Thus some functions of low level (e.g. $\text{HEAD}(l)$, $\text{TAIL}(l)$, $l$-list) and also the number of functions fixing semantic errors aren't defined. It is done to make the text of report shorter. Descriptions of functions fixing semantic errors are to be given in working out of NORMA translator's realisation.

2) The specification of the NORMA semantics isn't to be considered as the complete one. Some more details and modifications are possible to be added in the process of further researching. It may specify the rules of the NORMA language interpretation. Particularly the methods of specifying computational environment where Norma program can be carried out are under consideration now. The result of this research may give more details to the NORMA language.

3) Definition of an abstract machine $AM$-machine is a theoretical model which is the base for working out algorithms of Norma program translation. This problem isn't easy as the algorithms must be effective by time and memory, consider the peculiarities of the target computer and be equivalent to the notion of $AM$-machine. Obtained theoretical results (e.g. [3]) and worked out experimental version of the NORMA language's translator allow to consider this problem practically solvable.

4) Approach used in the semantics' specification is based on the representation of information dependencies between the variables from NORMA program by means of relations from relational algebra and the rules of interpreting these relations. This approach may be used for the specification of the semantics of the languages similar to NORMA which may be used for solving problems in other application domains, e.g. in the domain of relational data bases. To specify the semantics of the languages of considered type one may use other approaches based on the data driven computation model, e.g. data flow model.
References.

Russian Academy of Science
Keldysh Institute of Applied Mathematics

Andrianov A.N., Bugerya A.B., Efimkin K.N., Zadykhailo I.B.

NORMA

Language specification. Draft copy 1.22

Moscow 1996
Andrianov A.N., Bugerya A.B., Efimkin K.N., Zadykhailo I.B.


Abstract.

The NORMA language is a tool aimed at automatic solution of the mathematical physics problems on parallel computer systems.

The aim of the NORMA language is to eliminate the programming phase which is necessary to pass from computational formulae derived by an application specialist to a computer program. There is no essential difference between computational formulae and NORMA program structures - these formulae are an input for the NORMA translating system.

In fact NORMA program is a nonprocedural specification of problems to be solved. The mathematical problems connected with the synthesis of output program are solvable in the case of the NORMA language.

Draft specification of the NORMA language is given.
Contents

INTRODUCTION
1. Technical background
2. Aims
3. The goals of NORMA usage

NORMA LANGUAGE SPECIFICATION
1. Application domain and main characteristics
2. Syntax notation
3. Principal language elements
   3.1. Lexical rules
       3.1.1 Principal symbols
       3.1.2 Commentaries
       3.1.3 Tokens
       3.1.4 Identifiers
       3.1.5 Key words
       3.1.6 Constants
       3.1.7 Signs of operations
       3.1.8 Delimiters
3.2 Names
3.3 Basic data types
3.4 Expressions
   3.4.1 Arithmetical expressions
   3.4.2 Conditional expressions
4. Program structure
5. NORMA constructions
   5.1 Declarations
       5.1.1 Declarations of domains
           5.1.1.1 Declaration of unconditional domain
           5.1.1.2 Declaration of conditional domain
       5.1.2 Declaration of domain’s indexes
       5.1.3 Declaration of variables
       5.1.4 Declaration of index construction
       5.1.5 Declaration of domain’s parameters
       5.1.6 Declaration of input and output variables
       5.1.7 Declaration of external names
5.1.8 Declaration of distribution 's indexes
5.2 NORMA operators
5.2.1 Scalar operator
5.2.2 Assume operator
5.2.3 Function calls in NORMA
5.2.3.1 Standard arithmetical functions
5.2.3.2 Reduction functions
5.2.3.3 User's external functions
5.2.4 Part call
5.2.5 Interface with FORTRAN programs
5.2.6 Setting of sequential computing mode
5.3 Iteration

APPENDIX 1. Initial program representation.

BIBLIOGRAPHY

Introduction.

NORMA is a declarative language aimed at computation tasks' specifications. Translator of the language has besides traditional functions of semantic and syntax analysis a function of output program
synthesis during the translation. In other words the mode and the order of computations depended on target computer language and architecture are arranged.

The principal ideas of automated program design based on task’s specification were formulated by I.B. Zadykhailo [1] even in 1963. Their further development gave birth to NORMA and several versions of the translator [2-14]. Some examples of NORMA practical usage are described in [15,16].

Authors consider the approach used in Norma design and realisation useful for creation of new generation languages. The development of parallel computers enhances the value of this method.

This paper describes syntax and semantics of the NORMA language.

1. Technical background.

The idea of the NORMA language was produced by applied mathematicians from KIAM (Keldysh Institute of Applied mathematics) RAS (Russian Academy of Science). It was an attempt of automated program design based on the jobs (application of numerical methods to physical problems solution) prepared for further programming.

It was 1960 when the first works on this theme appeared in our institute (e.g., [17]). The method of writing jobs in this work was called “parameter record”. Later on such methods had the name “non-procedure specifications” and afterwards they were called “declarative specifications”. The languages used these methods are of specification type where the rules of data computations must be defined but the order of computations may be arbitrary (e.g. in parallel or sequential mode) according to the rules.

Many researches on programming theory and practical results proved the correctness of the chosen path. Some corroborating aspects are given below.

2. Aims.

Practical value of application software design depends on the degree of:
1) automation of design process,
2) portability and reliability of the output program,
3) convenience for the specialist in particular application domain.

The problems of providing all these items has been solving for the whole period of computer usage but there have been no universal and perfect methods for their solution yet. It can be explained by complicity of these problem and very fast development of the programming as a branch of science (new application domains and new hardware appear very quickly).

Progress in computer technology and architecture provides ample computational potentialities. But taking into consideration [1-3] requirements their perfect practical application has many difficulties. Thus when multiprocessor computers appeared the question of new application software design was raised.
Probably application programmer wants to know as little as possible about the architecture of a particular computer but to use its resources as much as he could. He'd like to have convenient tools for test and debugging but no problems with programs' portability. To work using generic methods and terms is also very important for application programmer. He needn't know questions which are alien to him [compiler's peculiarities, programming languages, system library organization and the questions of its effective usage, etc.].

The NORMA language and the approach to application program design is an attempt to take into consideration [1-3] requirements. The authors of this approach realize the difficulties of the fundamental task solution and the fact that universal and perfect way to this solution couldn't be found immediately. Though we can point out some goals of these approach even now.

3. The goals of NORMA usage.

High level of automation of the application design process.

Initial equations are formalized only to make input from keyboard easy. It is important that the issue of the formulae isn’t changed. Thus the programming phase is skipped and further design of the program is carried out automatically.

Reliability and portability of the programs.

If the method of the task solution is right and the formulae are written correctly an output program will be also correct (because the process of programming is automatic).

We have to underline that the strict specification of the task solution process on the application domain level is very important. This level is the most reliable because it deals only with the scheme of the computations and doesn't depend on the optimizations which can be realized during programming phase. It is evident that if we propose the language for the strict specification of the task solution process in the terms of application domain automatically designed program based on this specification will have the degree of reliability corresponding to the one of the translator and the translator’s optimization level.

NORMA used only the terms of application domain associated with the grid-based mathematical physics tasks solutions. Simplicity and reliability of the programs are provided by eliminating of the programming phase and required only the knowledge of application domain terms and their correct usage.

Synthesising translator from NORMA allows for computer architecture peculiarities. It provides portability of the application programs.
Method of parallel program design.

Generic mathematical formulae usage in NORMA has great possibilities for problem optimization in all cases including realization on parallel systems.

The specification of the task solution keeps its natural parallelism. This specification doesn't contain any adaptation constraints to computer architecture, programming language peculiarities or other requirements. It is "clear" specification of task solution which restricts only computation order according to the computation scheme content (informatic dependencies between variables). There is no term "memory" for storing different values in different periods of time. It simplifies natural parallelism revealing algorithms design and creation of the target program.

In the cases where computation order is important (e.g. it affects the precision of computations, rate of convergence, etc.) supplementary notions may be used (they are interpreted in a special way).

The development of the language makes its life longer. NORMA is rather young and capable.
Norma language specification.

1. Application domain and main characteristics.

The NORMA language is a specialized language applied to numerical-based specification of the mathematical physics problems solutions. First it was directed towards solving mathematical physics problems by means of grid difference method solutions. But later practice shows us that NORMA application domain could be more extensive.

Originally NORMA could be deciphered as: Non-procedure Specification of the Difference Algorithm Models. Nowadays we decode this abbreviation as NORMA1 level of computer - mathematician communication. Mathematician's formulae are input almost directly into the computer system.

NORMA requires no information about computation order and ways of computation process organization. The order of the sentences may be arbitrary (informatic connections are revealed during the translation).

Value can be assigned to any variable in NORMA only once. This feature characterizes the level of the NORMA language. There is no such terms as "memory", side effect, assignment statement, control operators in NORMA.

These characteristics and some other constraints (first on the form of index statements and ways of index domain specifications) substantiate solvability of output program synthesis problem [6,9]. Generic solution of this problem has substantial mathematical difficulties (the task may be NP-complete or undecidable). On the other hand researches on NORMA applications and design show that the constraints are acceptable practically [15,16].

2 Syntax notation.

Extended Backus - Naur form is used in the given syntax notation.

\( \{A\}^* \), \( \{A\}^+ \), \( \{A_1, \ldots, A_n\} \), \( \{A\} \) symbols mean

\[
\begin{align*}
\{A\}^* & ::= \emptyset | A | AA \ldots \\
\{A\}^+ & ::= A | AA \ldots 
\end{align*}
\]
\{ A_1, \ldots, A_n \} ::= A_1 \mid \ldots \mid A_n
\}
::= \emptyset \mid A

where A - any language object,
\emptyset - empty,
\mid - alternative choose,
\ldots \text{ etc.}

Syntax notions are written in italics, but words and symbols of common usage are printed in a usual way. As a rule, alternative constructions are arranged in columns (each construction on a separate line).

Sometimes half-underlined syntax constructions are used, e.g. \textit{name - set}. Syntactically this name is identical to symbol \textit{name} but underlined part of the construction has additional semantic information.

Symbol \textit{list-element} substitutes non-empty list of \textit{elements} enumerated by comma:
\begin{verbatim}
list-element :
element \{ .element \} *
\end{verbatim}
The definition of the element is given in every particular case.

Usually syntactical rules are given first; commentaries, semantic explanations, examples, etc. follow.

3. Principal language elements.

3.1 Lexical rules.
Symbols are a base for all the constructions, they are original language elements. The number of principal symbols is fixed and used for the design of any language constructions. The number of supplementary symbols isn’t fixed and settled by the type of computer equipment. They are used for symbolic constants formation of symbolic constants and data representation for medium.

3.1.1. Principal symbols.
\begin{verbatim}
principal-symbol :
letter
digit
special-symbol
\end{verbatim}
letter:
    { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z }

digit:
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
special-symbol:
    special-symbol-not-apostrophe
    apostrophe
special-symbol-not-apostrophe:
    { space, =, >, >, +, -,* /, (, ), [ , ], !, ?, ..., # }
apostrophe:

space:
    Space has no graphical representation.

3.1.2. Commentaries.
The line which begins from the sign "!" or the part of line which follows the sign "!" is a commentary.

3.1.3. Tokens.
There are 5 classes of tokens: identifiers, key-words, constants, operation signs, delimiters.
Spaces, LF, commentaries are considered tokens' delimiters and have no influence on the program semantics.

3.1.4. Identifiers.
identifier:
    letter { { digit, letter } }*

The letters of upper register differ from the ones of low register. Identifiers can have any length. The number of the meaningful symbols is defined in the specification of the translator's source language (usually first 6 symbols are meaningful).

3.1.5. Key words.
These key-words are used in the language:

```
MAIN PART    VARIABLE        ITERATION
PART         DEFINED ON     BOUNDARY
```
FUNCTION
BEGIN
END PART
RESULT
EXTERNAL FUNCTION
EXTERNAL PART
DISTRIBUTION INDEX
INDEX
REAL
INTEGER
DOUBLE
MACRO INDEX
DOMAIN
PARAMETERS
INPUT
OUTPUT
ON
END BOUNDARY
INITIAL
END INITIAL
EXIT WHEN
END ITERATION
FOR
ASSUME
COMPUTE

3.1.6. Constants.
constant :
    arithm-constant
    string
    arithm-constant:
        int-constant
        real-constant
        double-constant
    int-constant :
        \{ digit \}*
    real-constant :
        body \{ E \{ \{ +,- \} \} power \}
    double-constant :
        body \{ D \{ \{ +,- \} \} \} power \}
    body :
        int-constant . int-constant
    power
        int-constant
    string
        \{ \{ \{ symbol-not-apostrophe, \cdot \} \} \}**
    symbol-not-apostrophe:
        letter
digit
special- symbol-not-apostrophe
supplementary-symbol
The rules of writing constants are similar to the ones in the Fortran language.

The type and the value of the constant is defined by the way of its representation. These are examples of arithmetical constants:

\begin{verbatim}
101 999 0.1 1.0
10.5E-6 1.0E+7 1.0E7 0.1D-8 15.333D4
\end{verbatim}

These are examples of the strings:

'Hello, world!'

'Ía “ñi ñññ = ‘

3.1.7. Signs of operations.

\textbf{operation}:

\{ real-operation, log-operation, arithm-operation \}

\textbf{real-operation}:

\{ =,>,<,<=,>=,<,>,<< \}

\textbf{real-except-not-equal-operation}:

\{ =,>,<,>=,<= \}

\textbf{log-operation}:

\{ AND, OR, NOT \}

\textbf{arithm-operation}:

\{ +,-,\*,/,**, \}

Symbols $\geq$ and $\leq$ are used for comparisons $\geq$ and $\leq$, $<>$ and $>>$ symbols - for $\neq$.

3.1.8. Delimiters.

\textbf{delimiter}:

\{ space,\,().[,]\,?.\,\,...\,\# \}

3.2. Names.

\textbf{name}:

\textbf{identifier}

\textbf{list-name}:

\textbf{name} / \textbf{.name} */

Name is an identifier. It has a particular meaning and defines a particular program's object (e.g. variable, domain, part of the program). There can't be two definitions of the objects which have the same name.

The rule of name localization is true for every program element (part or function). The names declared in the program element are localized in it. There is no notion "global variable" in this language. The
names of the functions or the parts if they are actual or formal parameters of other parts or functions are indicated in the external names declaration EXTERNAL FUNCTION or EXTERNAL PART [see 5.16].

Half - underlined syntactical symbols e.g. name-set are syntactically equivalent to symbol name. Underlined part of the construction has additional semantic information.

Symbol list-name substitutes non-empty list of the names enumerated by comma.

3.3 Basic data types.

There are basic data types specified in the language:

integer
real
double
logical
string

Value from the set of integers is assigned to the element of integer data type. There are arithmetical and comparison operations for integer type elements.

Value from the set of real numbers is assigned to the element of real data type. There are arithmetical and comparison operations for real type elements.

Value from the set of real numbers is assigned to the element of double data type and it allows to use double precious real numbers. There are arithmetical and comparison operations for double type elements.

There are variables of integer, real and double type in the language.

Value "true" and "false" is assigned to the element of logical data type. It is used for definition of condition. There are logical operations for logical type elements. There are no variables of logical type in the language.

The element of string data type is a set of symbols from principal symbol set and supplementary symbol set. There are only constants but no variables and operations of this type.

3.4. Expressions.
3.4.1. Arithmetical expressions.

\[
\text{arithm-expression} : \\{ [ +, - ] \} \text{ term} \{ \text{ arithm-operation} \text{ term} \}^* \\
\text{term} : \\
\quad \text{arithm-constant} \\
\quad \text{call-function} \\
\quad \text{name-index} \\
\quad \text{name-domain-parameter} \\
\quad \text{name-scalar} \\
\quad \text{variable-on-domain} \\
\quad (\text{arithm-expression}) \\
\text{variable-on-domain} : \\
\quad \text{name-variable-on-domain} \{ [ \text{list-index-expression} ] \} \\
\text{const-expression} : \\
\quad \{ [ +,- ] \} \text{ const-expression-without-sign} \\
\text{const-expression-without-sign} : \\
\quad \text{integer-term} \{ \text{ arithm-operation} \text{ integer-term} \}^* \\
\text{integer-term} : \\
\quad \text{int-constant} \\
\quad \text{name-domain-parameter} \\
\quad (\text{ const-expression } ) \\
\text{index-expression} : \\
\quad \text{name-index} \{ [ +,- ] \} \text{ const-expression-without-sign } \\
\text{name-index} = \text{ const-expression-without-sign} \\
\text{name-index} = \text{name-index} \{ [ +,- ] \} \text{ const-expression-without-sign } \\
\text{name-index-construction}
\]

The result of the index expression computation in a fixed point \( T \) of the domain (5.1.1.) is coordinates of the point \( T \). These coordinates can be defined according to the rules:

1. all the names of the index constructions included into the index expression are substituted to the corresponding lists of index expressions from MACRO INDEX declaration:

2. indexes are substituted to the constant expressions, values of \( T \) coordinates (taking into account index displacement if it is given) or values of the other given indexes in the point \( T \).
For example, the list of index expressions \([\text{MyIJ}, k=7, l=i+4]\) for the point \(T\) with \(i=9, j=15, k=3, l=37\) (if we have the declaration MACRO INDEX MyIJ [i..1..]) gives us the value of the point \(T'\) from index domain with \(i=8, j=15, k=7, l=13\) coordinates.

The order of **arithmetical expression** computation is:

1. **index expressions** computation;
2. finding arguments (actual parameters) of the functions (including determination of reduction functions operation domain \([5.2.3]\));
3. carrying-out in-brackets (and) operations: if there are several brackets of the one level, they are examined from the left to the right;
4. carrying-out raising to the ** power operations: if there are several operations of the one level, they are examined from the right to the left;
5. carrying-out of multiplication and division * and / operations: if there are several operations of the one level, they are examined from the left to the right;
6. carrying-out of sum and difference + and - operations: if there are several operations of the one level, they are examined from the left to the right.

The type of arithmetical expression result is defined by the types of operation results. The type of operation result can be:

1. integer, if both operands are integer:
2. double, if even one operand is double:
3. real in any other cases.

**Constant expression** is a particular case of **arithmetical expression**. The examples of arithmetical expressions:

\[
\begin{align*}
\text{AL12}[J=2, M=1]+\text{STEPLP} \cdot JJ \\
Q+1.0D5/((R[Q-1]-R) \cdot (R[Q-1]-R[Q+1])) \\
\text{CN} \cdot \text{TRAP}(\text{STEP, } 2, \text{MV}+1, \text{MMR ON POLL})
\end{align*}
\]

3.4.2. **Conditional expressions**.

**log-expression**:

\[
[ \text{NOT} ] \text{logic-term} \ [ \text{log-operation logic-term} ]^a
\]

**logic-term**:

- **comparison**
  - ( **log-expression** )

**comparison**:

- **arithm-expression** **comp-operation** **arithm-expression**
condition-on-domain:
  log-expression
condition-on-index:
  name-index rel-except-not-equal-operation
    name-index [ \{,,-\} const-expression-without-sign ]
    name-index rel-except-not-equal-operation const-expression-without-sign

The order of operations for logical expression computation:
(1) carrying out of arithmetical expressions computation;
(2) carrying out of in-brackets ( and ) operations; if there are several brackets of the one level, they are examined from the left to the right.
(3) carrying out of comparisons =,>,<,>,<=,>=,<> ; if there are several operations of the one level, they are examined from the left to the right.
(4) carrying out of logical negation NOT operations; if there are several operations of the one level, they are examined from the left to the right.
(5) carrying out of logical multiplication AND operations; if there are several operations of the one level, they are examined from the left to the right.
(6) carrying out of logical sum OR operations; if there are several operations of the one level, they are examined from the left to the right.

There are examples of logical expressions:
ABS(MOD-X1+Y1)<=1.0D-20
NOT (II=(2*N+1)*(2*NV+1)) AND NU=0
I=J+1

4. Program structure.

program:
  {part}^*
part:
  main-part
  simple-part
  part-function
main-part:
  MAIN PART name-part declaration-of-part
simple-part:
PART name-simple-part . declaration-of-part

part-function :

FUNCTION name-function [type-function] . declaration-of-function

declaration-of-part :

formal-parameters-of-part BEGIN body-of-part END PART
formal-parameters-of-part :

list-name [RESULT list-name]

declaration-of-function :

formal-parameters-of-function BEGIN body-of-part END PART
formal-parameters-of-function :

list-name

body-of-part :

[element-of-part]*

element-of-part :

declaration .

operator .

iteration .

Program in NORMA consists of one or several parts. The parts may be of three types - main part, simple part and part-function (corresponding to key-words MAIN PART, PART, FUNCTION). Parts can call each other by name and communicate data with help of formal and actual parameters or through external files with help of INPUT and OUTPUT declarations.

Localization rule is true for each part: the names declared in the part are localized in it. There are no global variables in the language.

The main part must be in the NORMA program and be the only one (it has no formal parameters. Main part calls and recursive calls are prohibited.

There is name of the part, text-commentary (if you need) and the list of formal parameters in the part header. Formal parameters must be declared in the body of part with help of declaration on domain, scalar declaration, domain’s parameters declaration and declaration of external.

Parameters-variables declared before the key-word RESULT in the list of formal parameters are initial data for computations specified in the part, the parameters declared after are the results of computations. One and the same parameter can’t be initial and result at the same time (it causes reassignment which is prohibited in NORMA). The key-word RESULT isn’t used in part-function (the result of computations is connected with the name and the type of function).

There are definitions, operators and iterations in the body of part. In general their order is arbitrary, possible constraints are defined in translator's source language description).
It is an example of main part header:

**MAIN PART** Linear. ! Variant of May 25 1992:

! linear approximation =========
! the case of axial symmetry for M=1:
! computation is being done on Beta angle for the point I=0:
! further for I=1,2,N values
! are being extended on symmetry for V,FL,DAVL.

**BEGIN**

body of part

**END PART**

It is an example of simple part header:

**PART** IntFKP.

! Computation of integral from FKP function
! with interpolation of integral function
  BETA,ALPHA, ! node points
  ALL,ALC,ALR, ! points for interpolation
  STEP, ! integration step
  FKP, ! name of integral function
  ! result IntResult

**RESULT** IntResult

**BEGIN**

**EXTERNAL FUNCTION** FKP DOUBLE.

**VARIABLE** ALPHA,BETA,ALL,ALC,ALR,STEP,IntResult DOUBLE.

body of part

**END PART**

5. NORMA constructions.

We can divide Norma constructions into *declarations*, defining program objects (e.g. domains, indexes, variables) and *constructions* for computation rules specification.

5.1 *Declarations*

*declaration:*

declaration-of-domain

declaration-of-domain-indexes

declaration-of-scalar-variables

declaration-of-variables-on-domains

declaration-of-index-constructions

declaration-of-distribution-indexes
Objects which can be declared in NORMA are: domains, scalar variables (scalars), variables specified on grid, index constructions, distribution indexes, domain parameters, input and output variables, names of external functions and parts.

5.1.1 Declarations of domains.

domain :
    new-domain-without-name
    name-domain

unconditional-domain :
    new-domain-without-name
    name-unconditional-domain

name-domain :
    name-unconditional-domain
    name-conditional-domain

name-unconditional-domain :
    name-rectangular-domain
    name-diagonal-domain

The notion domain was introduced in NORMA for index space representation. Domain is a complex of integers numbers sets \( \{i_1, \ldots, i_n\}, n>0, i_j>0, j=1,\ldots,n \). Each set gives coordinates of the point from \( n \)-dimensional space. Unique name - index name is connected with one of the directions (coordinate axes) of the \( n \)-dimensional space.

The domain defines coordinates values of index space points, but not the values of the calculated variables in these points. For example if you need to calculate the value of the variable \( Y_{ij} \), \( i,j=1,\ldots,n \) on some grid \( X_{ij}, i,j=1,\ldots,n \), which was introduced during task solution (e.g. by formula \( X_{ij} = F(h, i, j) \), \( F \)-given function, \( h \) - given parameter) you should:
(1) define the domain consists of points \(i,j = 1,...,n\);

(2) define variables \(X\) and \(Y\);

(3) set the rule of grid \(X_{ij}\) computation : \(X_{ij} = F(h,i,j)\) and the rule of \(Y_{ij}\) values computation \(Y_{ij} = G(X_{ij})\) (\(F,G,h\) are considered to be given somehow).

Domain in NORMA can have a name. The operations of modification and multiplication are defined on the domains. Domain indexes aren't specially declared, they are introduced in domains' declaration. Domain can be conditional and unconditional. Conditional domain consists of the index space points. The number and the coordinates of these points can change depending on fulfilment or non-fulfilment of conditions on domain. Unconditional domain consists of the points from index space, which number and coordinates may be defined during the translation.

There are two different terms in NORMA declaration of domain (named conditional or unconditional domain) and use of domain (syntactically it is the name of domain or new domain without name). Domains are used in declarations of the variables declared on domain, setting computation domain in ASSUME operators, input and output variables declarations, setting domains of actual parameters in part or functions calls in reduction functions.

5.1.1.1 Declaration of unconditional domain.

\[
\text{declaration-of-rectangular-domain :}
\]

\[
\text{multidimensional-domain}
\]

\[
\text{new-domain}
\]

\[
\text{multidimensional-domain :}
\]

\[
\text{one-dimensional-domain}
\]

\[
\text{name-multidimensional-domain : } \{ \text{domain-product} \}
\]

\[
\text{domain-product :}
\]

\[
\text{component-domain [ ;component-domain ]}^* \\
\text{component-domain :}
\]

\[
\text{multidimensional-domain}
\]

\[
\text{name-unconditional-domain}
\]

\[
\text{one-dimensional-domain :}
\]

\[
\text{name-one-dimensional-domain : } \{ \text{name-index=value} \}
\]

\[
\text{value :}
\]

\[
\text{range}
\]

\[
\text{const-expression}
\]

\[
\text{range :}
\]
const-expression .. const-expression

new-domain:
   [ name-new-domain: ] new-domain-without-name

new-domain-without-name:
   name-unconditional-domain /list-modification

modification:
   name-index=value

name-onedimensional-domain { ( +,- ) boundary-function}+

boundary-function:
   LEFT(const-expression)
   RIGHT(const-expression)

name-rectangular-domain:
   name-onedimensional-domain
   name-multidimensional-domain
   name-new-domain

description-of-diagonal-domain:
   name-diagonal-domain: name-unconditional-domain /list-condition-on-index

The notion of one-dimensional domain is a key notion in rectangular dimensions declaration. One-dimensional domain is used for setting the range of points on some coordinate axe in the index space. In the simplest case the name of one-dimensional domain, index name and the boundaries of index values changing are declared in one-dimensional domain declaration.

RegionK: (k =1..15).

Name-onedimensional-domain can be used for reference to this domain.

The name of index is an index variable. The set of its values is defined by the range const-expression .. const-expression. The boundaries of the range are integer positive constant expressions built from integer constants, domain’s parameters and arithmetical operations. Particular values must be assigned to the domain’s parameters in the declaration of domain’s parameters. The value of the low boundary mustn’t be more than the value of upper boundary. For example,

RegionK: (k =1..15).

Multidimensional domain is built by operation “;” of rectangular domains multiplication.

It is an example of two-dimensional domain declaration. This domain is built by the multiplication of one-dimensional domains **AxisK** and **AxisL**:

Square: (AxisK: (k=1..15) ; AxisL: (l=1...5)).

Domain **Square** may be defined in different ways ( e.g. taking into account the previous definition ):

Square: (RegionK; AxisL: (l=1...5)).
The "." operation of rectangular domains' multiplication has an ability:

\[ A;B = B;A \quad \text{if A and B are domains.} \]

It means that the order of the index space directions in domain's declaration isn't fixed (or fixed arbitrarily, that is the same for user). If the order of index space directions is important (e.g. consistency of directions is needed for using the same variables on the same domains in the different parts) it may be set by INDEX declaration (5.1.2).

If the domains are operands of domains' multiplication operation they must have different index names.

The issue of rectangular domain modification may be in addition of some points, deletion of the points or changing the range. The modification of the first two types is defined by the boundary functions \texttt{LEFT}(n) and \texttt{RIGHT}(n). Function \texttt{LEFT} is applied to the left boundary of the range and function \texttt{RIGHT} - to the right one. Sign "+" means that the points are added to the one-dimensional domain, sign "-" means that they are deleted. Both functions have one parameter \( n \) which defines the number of the points. Integer positive constant may be an actual parameter of the function only. Call to the functions is allowed only in the context with the name of one-dimensional domain setting modified range:

\[ \text{name-one dimensional domain +LEFT}(n) \]

or

\[ \text{name-one dimensional domain +RIGHT}(n). \]

E.g. the declaration

\[ \text{FlushK: Square/AxisK-LEFT(2)+RIGHT(2).} \]

defines domain \text{FlushK} which consists of points \( k=3\ldots17 \).

The boundaries of the resulting range (after using \texttt{LEFT} and \texttt{RIGHT} functions) must be positive integer and the left boundary mustn't exceed the right one.

Declaration

\[ \text{Square2: Square / AxisK+Left(1)-Right(2).} \]

isn't right because the left boundary of the resulting range \( k=0\ldots13 \) isn't positive.

You can change component one-dimensional domain by the explicit redeclaration of the ranges. For this purpose index name of direction and its new value must be declared in the modification.

E.g.

\[ \text{Newsquare: Square / AxisK+RIGHT(3), L=50...80.} \]

If the list of modifications has more than one element modifications declared in that list act on modified domain in writing order (from the left to the right).

Besides rectangular domain declarations in NORMA there is a possibility of diagonal domain settings by the superposition of conditions on previously declared domain (rectangular or diagonal).

Diagonal domain is defined by conditions on indexes of some previously declared domain \( D \). It consists of domain \( D \) points in which the conditions are true. E.g.

\[ K:\{ (k=1\ldots10) ; (l=1\ldots10) \} \quad \text{Diagonal: } KL/ k=1. \]
set domain **Diagonal** consists of the points \((k=1, l=1), (k=2, l=2), \ldots, (k=10, l=10)\).

Indexes on variables used in the notation of conditions on indexes can be either internal (from **D** domains set of indexes) or external (5.2.3.2).

The valid form of conditions on indexes defines statically the points of diagonal domain (it is also possible for rectangular domain). That’s rectangular and diagonal domain are **static** objects. They differ from **conditional** domain described below.

**5.1.1.2 Declaration of conditional domain.**

*declaration-of-conditional-domain :

```
name-conditional-domain, name-conditional-domain:
```

```
name-domain / condition-on-domain
```

Besides static domains in NORMA you can set conditional domain consists of the points from index space. Their number and coordinates can change depending on fulfilment or nonfulfilment of conditions on domain.

Here we describe how you can set conditional domain. Previously declared domain **D** is divided into two disjoint subdomains **D1** and **D2**. The first subdomain consists of the points where the conditions on domain are true, the second one - where they are false. Besides \[ D1 \cup D2 = D1, D1 \cap D2 = \emptyset \].

E.g. set domains:

**Domain**:

```
( ( i=2..m ) ; ( j=1..m ) ) .
```

```
Domain1,Domain2: Domain / x+y[ i-1..j ] \cdot \neg z[ j+1 ] > 0 .
```

This declaration defines division of initial domain **Domain** into domains **Domain1** and **Domain2** and

```
Domain1 \cup Domain2 = Domain, Domain1 \cap Domain2 = \emptyset .
```

**Domain1** consists of the points of **Domain** in which the condition \( x + y[ i-1..j ] \cdot \neg z[ j+1 ] > 0 \) is **true**.

**Domain2** - where the condition is false.

Indexes of the variables used in the conditions notation can be either internal (from **D** domains set of indexes) or external (5.2.3.2).

Program fragment defining conditional domains **B2PI** and **BNot2PI**, and their further division into smaller domains (first domain **BNot2PI** is divided into two conditional domains **Nodes** and **NotNodes**, and then domain **NotNodes** is divided into two conditional domains **DomainTrue**, **DomainFalse**) is given below:

```
DOMAIN PARAMETERS N=3, NV=3 .
Bnu: (N=0..2*N),BF: (I=0..2*N+1) \cdot (2*NV+1). Bnu: (Bnu) .
B2PI,BNot2PI: Bnu /I=0..2*N+1*(2*NV+1) AND NU=0.
Nodes,NotNodes: BNot2PI /ABS(BTNodes-BT) < 0.0001.
DomainTrue,DomainFalse: Nodes /ABS(BTnu - BTNodes) < 0.0001.
VARIABLE BT, BTNodes DEFINED ON BT DOUBLE .
VARIABLE BTnu DEFINED ON Bnu DOUBLE .
```
5.1.2 Declaration of domains' indexes.

declaration-of-domain-indexes:

INDEX list-name-index

The order of the index space directions in domain's declaration isn't fixed (or fixed arbitrarily, that is the same for user). If the order of index space directions is important (e.g. consistency of directions is needed for using the same variables on the same domains in the different parts) it may be set by declaration of domains' indexes. The order of index space directions is the same as the order of index's names enumeration in declaration INDEX (from the left to the right).

We consider call of part B from part A with variable X values communication from A to B. Variable X is defined on SquareInA domain:

PART A.
BEGIN
INDEX k,l.
SquareInA: (AxisK: (k=1..5) ; AxisL: (l=1..10) ).
VARIABLE x DEFINED ON SquareInA.
COMPUTE B( X ON SquareInA).
END PART

PART B. X
BEGIN
INDEX i,j.
SquareInB: (AxisK: (i=1..5) ; AxisL: (j=1..10) ).
VARIABLE x DEFINED ON SquareInB.
END PART

Declarations INDEX k,l and INDEX i,j make correspondence between indexes: i~k,j~l. In other words the values of X variable are considered in the one and the same way in both parts. If INDEX i,j declaration from part B is substituted to INDEX i,j the values on corresponding directions will be inconsistent: 5 ≠ 10, 10 ≠ 5.

5.1.3 Declaration of variables.

declaration-of-scalar-variables:

VARIABLE list-name-scalar [ type ]
declaration-of-variables-on-domains:

VARIABLE list-definition-variables-on-domain [ type ]

definition-variables-on-domain:

list-name-variable-on-domain

DEFINED ON unconditional-domain

type:

(REAL, INTEGER, DOUBLE)

Scalar variables and variables on domain are arithmetical variables. There is a unique name and the type of variable in the declaration corresponding to every arithmetical variable. The types of variables are REAL, INTEGER or DOUBLE (default REAL).

It is an example of scalar 's declaration:

VARIABLE Alpha, X, Y, VARIABLE IJK, Z INTEGER.

Every variable on domain is connected with the domain in the declaration. This domain defines index's names, which can be used in index expressions in calls to this variable (the order of index expressions isn't important). There is no special declaration for indexes, they are introduced in domain's declarations.

Square: (AxisK: (k=1..5) ; AxisL: (l=1..5)).

VARIABLE First, Last DEFINED ON Square,

SingleK DEFINED ON AxisK. SingleL DEFINED ON AxisL.

Given declarations define variables First, Last on domain Square. It means that these variables can have corresponding indexes k and l in the index expressions. Variables SingleK and SingleL are defined on domains AxisK and AxisL. It means that they can have corresponding indexes k and l in the index expressions. In this call First(k=1..1) and First(k=1..k-1) are equivalent (5.1.2).

In call to variable on domain the rule of default indexes setting is performed: index expressions which coincide with index name may be omitted. E.g. notations First(k), First(k), First(l), First for variable First are equivalent.

If some index is set by constant (constant expression) you must declare which index direction this constant concerns to, e.g. First(k=5..1). If you need to link two directions by some formula you must do it in explicit way, e.g. diagonal elements of the matrix First may be defined as First(k=k) or First(l=k).

5.1.4. Declaration of index construction.

declaration-of-index-construction:

MACRO INDEX name-index-construction [ list-explicit-index-expression ]

explicit-index-expression:

name-index [ [ +,- ] const-expression ]
name-index = const-expression

name-index = name-index [ { +,- } const-expression ]

Index construction is used for shortening notation of complicated index expressions. It is the simplest case of macrodeclaration. e.g. here are the declarations of index constructions Short and SecondShort and the declaration of domain IJKL and variable V.

IJKL : ( i=1..10 ) ; ( j=1..10 ) ; ( k=3..17 ) ; ( l=3..18 ) )

VARIABLE v DEFINED ON IJKL DOUBLE.

MACRO INDEX Short [ i-1, j-1, k+1 ]

MACRO INDEX SecondShort [ i-1, j-1, k+1, l+2 ].

This declarations allow notations of v[Short, l-2] or v[l-2,Short] type for setting variable v[ i-1,l-2,
j-1, k+1 ] and v[ SecondShort ] for setting variable v[ i-1, l+2, j-1, k+1 ].

5.1.6. Declaration of domain parameters.

declaratio-of-domain-parameters :

DOMAIN PARAMETERS list-prescription

prescription :

name-domain-parameter=int-const

Range boundaries in domain
's declaration may be set by parameters of domain ( in implicit way ). The values of these parameters must be defined in the part. in the domain parameters declaration. E.g.

DOMAIN PARAMETERS M=3, J=90, DomainLeftBoundParameter=15.

Parameters of domain may be included into arithmetical expressions and constant expressions.

5.1.6 Declaration of input and output parameters.

declaratio-of-input :

INPUT list-input-scalar

INPUT list-input-on-domain

input-scalar :

name-scalar [ attributes ]

input-on-domain :

name-variable-on-domain

declaratio-of-output :

OUTPUT list-output-scalar

OUTPUT list-outputs-on-domain

output-scalar :

name-scalar [ attributes ]
outputs-on-domain:
    list-output-on-domain [ attributes ] ON domain
output-on-domain:
    name-variable-on-domain
attributes:
    ( list-attribute )
attribute:
    STR( int-constant )
    TAB( int-constant )
    SPACE( int-constant )
    string
    ORDER(list-name-index)
    LENGTH= int-constant
    FILE= 'file-name'
    ALL
    format
format:
    I int-constant
    ( F.E.D ) int-constant . int-constant

Declaration of input (output) variables means that all the variables mentioned in the list of input (output) are liable to input (output). E.g. declarations

B1,B2 : BZ<Eps.

INPUT Velocity ON A. OUTPUT Tau ON B1.

INPUT X, ALPHA.

are requests for input of scalar X, ALPHA values , value of Velocity in all the points of B1 domain where condition Z<Eps is fulfilled.

The order of variables input (output) isn't set, it is defined during the translation .Minimal piece of information which is input (output) as a whole is scalar or variable on domain. Attributes may control input (output) of these elements.

Attributes act on the variable (list of variables) declared before. E.g. declaration

INPUT R1.R2 (FILE ="myfile") ON Grid.

is a request for input of variables R1.R2 values on Grid domain ( in other words for all the index values of this domain) from file myfile.data.

Attributes help to control the form of input and output data and bind data with input and output files:

Ø STR(n) sets the skip of n-1 lines;
\( ^{i}\text{TAB}(n) \) sets \( n \) spaces before the beginning of the line;
\( ^{i}\text{SPACE}(n) \) sets \( n \) spaces in the line begging of the current position;
\( ^{i}\text{ 'string' } \) set output of the textual constant \( \text{string} \);
\( ^{i}\text{ORDER} \) defines the order of index changing for input; the highest index is the first from the right, the lowest is the first from the left; for one value of every high index the low index has run over all their values;
\( ^{i}\text{LENGTH} \) sets the length of record in the output file;
\( ^{i}\text{FILE} \) sets the name of input or output files; the method of setting file name is defined during realization (if you use translator version for personal computers input file has a default name with .dat extension; when you set file name this extension may be omitted; if attribute \( \text{FILE} \) isn’t given input is carried out from file \( \text{norma.dat} \) and output to display screen);
\( ^{i}\text{ALL} \) sets output variable’s name and its coordinates (index values) for every value of variable on domain.

\( ^{i}\text{format} \) sets format for numerical values for input and output and corresponds to format specifiers I, F, E, D in the FORTRAN language. There are default formats: E>15.8 for variables of \( \text{REAL} \) type, D15.8 for variables of \( \text{DOUBLE} \) type I5 for variables of \( \text{INTEGER} \) type.

For input variables only attributes of format and \( \text{FILE} \) type are allowed.

Here is an example of input variables declaration:

\[
\text{INPUT One (FILE='file1',F10.2) ON Grid1,}
\text{Two(FILE='file2') ON Grid2.}
\]

Here are examples of output variables declarations:

\[
\text{OUTPUT Velocity(STR(5),TAB(7), 'Velocity = \text{'},F9.1\text{'}) .}
\text{OUTPUT X(FILE='FILE1', ORDER(J,K,I), ALL, IS.1) ON Grid.}
\text{OUTPUT Y('Matrix',SPACE(10), 'of values',SPACE(10), 'Y parameter',}
\text{FILE='OTS', ALL,F15.2) ON ABC.}
\]

For output in more complicated form (diagrams, tables, etc.) you should use standard libraries and packets tools or your own programs written in other languages.(5.2.5).

Syntax of input data files:

\[
\text{file:}
\text{input-element \{input-element \}^*}
\text{input-element:}
\text{name=scalar = arithm-constant;}
\text{name=variable-on-domain\{list-index-range\} = data;}
\text{index-range:}
\text{name-index = int-constant .. int-constant}
\]
data:
  list-data-element
data-element:
  arithm-constant
  int-constant(data)

Programmer can:
✓ place input elements in any order,
✓ control the order of numerical values in file by changing index order,
✓ not keep to one format in numerical values notation,
✓ write repeated data in a short form.

Here is an example of input data file's contents:
C(K=1..10)=5(-1.1),5(1.01);
ALPHA=3.17; BETA=-.012; GAMMA=1.0E-10
C(K=11..20)=5(10.1,5(-1.01));

Here is an example of input data file data.dat contents for Gauss program:

MAIN PART Gauss.

Solution of linear equations by Gauss-Jourdan method.

BEGIN
Ot:(t=0..n). Oj:(i=1..n). Oji:(j=L..n).
Otij:Ot:Otij / t=1..n. Ot1:Ot1 / t=1..n.

DOMAIN PARAMETERS n=5.

VARIABLE a ON Oij. VARIABLE m ON Oti.

VARIABLE b, ON Oi. VARIABLE r ON Oi.

INPUT a(FILE='data') ON Oij, b(FILE='data') ON Oi.

OUTPUT x(FILE='results',ALL) ON Oi.

FOR Otij[t=0] ASSUME m=a.
FOR Ot / t=0 ASSUME r=b.
OtiEQt1j>OtiNEQt1j:Otij / int. OiEQt1i>OiNEQt1i:Ot1i / int.
FOR OiEQt1j ASSUME m = m[t-1.i=t]/m[t-1.i=t].
FOR OiEQt1j ASSUME r = r[t-1.i=t]/m[t-1.i=t].
FOR OiEQt1j ASSUME m = m[t-1,i=t]/m[t-1,i=t].
FOR OiEQt1j ASSUME r = r[t-1,i=t].
FOR Oi EQt1j ASSUME x = r / t=n.

END PART.

Input file data.dat:
B(t=1..3)=5.0, 13.0, 3.0;
A(t=1..5,i=1..5)=2.0, 3.0, -4.0, 5.0, -1.0, 3.0, 4.0, -1.0, 6.0, 1.0,
The values of data multidimensional arrays are placed in the file corresponding to the given indexes. The values of the first index from the right are changed the first: in the given example matrix A(I,J) is set on rows, in other words I=1, J=1,2,3,4,5, then I=2, J=1,2,3,4,5 etc. till I=5, J=1,2,3,4,5.

The way of input file data.dat setting isn't the only one.

5.1.7 Declaration of external names.

\textit{declaration-of-external-names}:
- \textit{declaration-of-external-functions}
- \textit{declaration-of-external-parts}

\textit{declaration-of-external-functions}:

\textbf{EXTERNAL FUNCTION} \textit{list-name-external-function} [ \textit{type} ]

\textit{declaration-of-external-parts}:

\textbf{EXTERNAL PART} \textit{list-name-external-simple-part}

If the names of the functions or parts are their actual or formal parameters they are indicated in the \textit{declaration of external names} by all means. Default type of external function is \textbf{REAL}.

\textbf{EXTERNAL FUNCTION} \textit{Last,First} \textbf{DOUBLE}.

\textbf{EXTERNAL PART} \textit{Middle}.

5.1.8. Declaration of distribution indexes.

\textit{declaration-of-distribution-indexes}:

\textbf{DISTRIBUTION INDEX} \textit{name-index} = \textit{simple-range} [ \textit{name-index} = \textit{simple-range} ]

\textit{simple-range}:

\textit{int-constant} [ .. \textit{int-constant} ]

Declaration of distribution indexes is used for representation of two index directions from task domain index space on processor element (PE) matrix of distributed system. If functions and parts have this declaration they are called distributed, if there is no such declaration in their content they are called nondistributed systems. You can meet this declaration in the distributed part or function only once; it is prohibited in the main part in other words the main part is always nondistributed.

Given declaration results in data and control distribution between PE elements of the system or automatic generation of data exchange between PE if you need it. The subject of distribution is variables which indexes coincided with ones in declaration \textbf{DISTRIBUTION INDEX}. E.g. if there is a declaration
DISTRIBUTION INDEX \( i=2..8, j=11 \).

All the variables defined on the domains with \( i \) and \( j \) indexes will be distributed on \( \text{PE}_{i,j} \) with virtual row numbers \( i=2..8 \) in the column number \( j=11 \) of \( \text{PE} \) matrix.

Computations defined in nondistributed part or function are carried out as a whole in one \( \text{PE} \) (though there may be several such \( \text{PEs} \).)

Declaration

DISTRIBUTION INDEX \( i=2..8, j=0 \).

Is false; the elements of \( \text{PE} \) matrix are considered to be numbered from 1.

5.2 Operators in NORMA.

Operator:

- scalar-operator

Operator-ASSUME

Call-part

There are three defined types of operators in NORMA. They are scalar operator, operator ASSUME and call part. The operators are used for description of computations for task solution.

5.2.1 Scalar operator.

Scalar-operator:

- name-scalar = scalar-arithm-expression

Scalar-arithm-expression:

- arithm-expression

Scalar operator is used for computation of scalar arithmetical values. In fact it is an analogue of assignment statement in traditional programming languages. Name of scalar is indicated in the left part of the operator and scalar arithmetical expression built in usual way from scalars, arithmetical constants, domain’s parameters, function calls, variables on domain with index-constants.

Variables defined on domain which index expressions are not constants can’t be included into scalar arithmetical expression (exception from this rule is arguments of reduction functions (5.2.3)).

\[ \text{ij: } i=1..\text{MaxI}, j=1..\text{MaxJ} \]  

Domain parameters \( \text{MaxI} = 3, \text{MaxJ} = 90 \).

Variable ScalarV INTEGER. Variable PI DOUBLE.

Variable ArrayV DEFINED ON IJ DOUBLE.

ScalarV = MaxI*(MaxJ-1)/2+SQRT(PI)/SIN(ArrayV[i=1,j=55]).
Reassignment in scalar operator is prohibited, that’s the next operator is false:
\[ \text{ALPHA} = \text{ALPHA}-1. \]

Section 5.2.2: Assume Operator

**Operator-Assume**

FOR domain **Assume** relation('; relation')* 

relation:

- name-variable-on-domain = arithm-expression 

**Call-part**

Assume operator is used for computation of arithmetical variables defined on domains.

The case of part call in the body of **Assume** operator is described in 5.2.4.

The terms of arithmetical expression in the right part of relation may be variables defined on domain, scalars, arithmetical constants, domain’s parameters, function calls, indexes.

Informally, **Assume** operator’s semantics is defined in the way described below. Let’s consider given relation

FOR \( D(i_1, \ldots, i_n) \) **Assume**

\[ X_{\text{IndL}(i_1, \ldots, i_p)}^{i_1} = F(X_{\text{IndR}(i_1, \ldots, i_q)}^{i_2}, \ldots, X_{\text{IndR}(i_1, \ldots, i_r)}^{i_k}, \text{Other}) \]

where

- \( D(i_1, \ldots, i_n) \) - **Assume** operator’s domain.
- \( (i_1, \ldots, i_n) \) - \( D \) domain indexes,
- \( X_{i_q}^{i_p}, 1 \leq q \leq k \) - variables names defined on domain,
- \( \text{IndL}(i_1, \ldots, i_p), 1 \leq p \leq n \) - index expressions of the left part,
- \( \text{IndR}(i_1, \ldots, i_q) \) - index expressions of the right part,
- \( F \) - function computed in the right part,
- \( \text{Other} \) - other terms of the right part.

Every relation set the rule \( F \) of variable \( X^i \) from the left part values computation based on the values of \( X_{i_1}^{i_1}, \ldots, X_{i_k}^{i_k} \) variables and the terms \( \text{Other} \) from the right part:

1. all the points \( (a_1, \ldots, a_n) \in D(i_1, \ldots, i_n) \) are defined;
2. value of \( X^i \) from the left part is computed in \( \text{IndL}(a_1, \ldots, a_n) \) point for every point \( (a_1, \ldots, a_n) \in D; \)
(3) index expressions \( IndR^q_i(a_1, \ldots, a_n) \) of all the variables \( X^i_k \), \( 1 \leq q \leq k \) from the right part of relation are computed for every point \( (a_1, \ldots, a_n) \in D \) and the set of right part arguments is defined
\[
X(a_1, \ldots, a_n) = \{X^i_k \mid IndR^q_i(a_1, \ldots, a_n), \ Other\}, \ 1 \leq q \leq k
\]

(4) if for some period of time for point \( (a_1, \ldots, a_n) \in D \) all the arguments from \( X \) have been computed the computation of \( X^l_{\text{indL}(a_1, \ldots, a_n)} \) value from the left part is possible, but if all the arguments haven't been defined the computation in that point in that period of time is impossible (it doesn't mean that it is impossible in all cases).

The name of ASSUME operator underlines the fact that it defines the rule of value computation in the only way but doesn't require immediate computation in particular place of program and doesn't arrange the order and the method (parallel or sequential) of computation.

E.g. if the variable is defined as
Matrix: \( (I=1..5) ; (J=1..5) \).

**VARIABLE DEFINED ON** Matrix.

then operator

**FOR** Matrix ASSUME \( X = 0 \).

is a request for 25 elements of variable \( X \) zeroisement. The method of this request realization isn't fixed in this language.

**NORMA** is a single-assignment language, reassignment is prohibited. That's further operators are incorrect:

**FOR** Matrix ASSUME \( X = Y ; X = Z \).

**FOR** Matrix ASSUME \( X = X+1 \).

Constraints on the index form the left part of relation (these are indexes without displacement) are not essential because domain \( D \) from the operator's header allows to define rather complicated relations. If you need to define the computation of
\[
X_{i,j} = F(Y_{i,j}) \text{ on domain } O_x : (I=1..N) ; (J=1..M)
\]
type you may do it in this way:

**OxH**: \( Ox/J=I \).

**FOR** OxH ASSUME \( X = F(Y,Z) \).

As we mentioned earlier functions may be included into arithmetical expression and scalar arithmetical expression.

5.2.3 Function calls in Norma.

**call-function** :

**call-reduction-function**
call-standard-function

call-external-function

call-reduction-function:

name-reduction-function

( ( name-domain ) arithm-expression )

call-standard-function:

name-standard-function( arithm-expression )

name-reduction-function:

{ SUM, MULT, MAX, MIN }

call-external-function:

name-external-function( ( list-in-parameter ) )

in-parameter:

arithm-expression

name-variable-on-domain ON domain-of-parameter

iterated-variable-on-domain ON domain-of-parameter

name-external-simple-part

name-external-function

domain-of-parameter:

name-unconditional-domain

name-unconditional-domain / index-expression

name-unconditional-domain / (list-index-expression)

iterated-variable-on-domain:

name-variable-on-domain [ name-iteration-index [-1] ]

There are defined standard arithmetical functions, reduction functions and external user's functions in the language.

5.2.3.1 Standard arithmetical functions.

Functions: sqrt(x), abs(x), esp(x), alog(x), alog10(x), sin(x), cos(x), asin(x), atan(x), entier(x) are standard functions.

5.2.3.2 Reduction functions

Functions: SUM (sum), MULT (multiplication), MAX (maximum), MIN (minimum) are reduction functions. Call to these functions:

name-function( ( name-domain ) arithm-expression).
Domain defines the number of domain's point where the function will be computed, arithmetical expression defines the number of values which the function are applied to. Here is an example:

\[ V_i = W_i + \sum_{j=1}^{n} A_{i,j} \cdot X_j, \quad i = 1, \ldots, n. \]

The fragment of this computation in NORMA:

```
Grid: (Oi: (1..N); Oj: (J=1..M)).

VARIABLE A DEFINED ON Grid.
VARIABLE V,W DEFINED ON Oi. VARIABLE X DEFINED ON Oj.

FOR Oi ASSUME V = W + SUM( (OJ) A*X).
```

Here is an example of embedded reduction functions:

```
SurfaceMax:
  (Left: (Three:(IV=1..3); A:(J=0..2*M); B:(I=0..2*N)));
  Right: (I=1..9); (W=0..2*M); BNU:(NU=0..2*N)).
SUMPOL:(W=0..2*M); BNU).
ThreeAB:(Three:A:B); OLINE:(LINE=1..3).
VARIABLE VV,V DEFINED ON ThreeAB DOUBLE.
VARIABLE JSEQ DEFINED ON SurfaceMax DOUBLE.

FOR ThreeAB ASSUME
  VV=SUM( (SUMPOL)
  SUM((OLINE) V[IV=LINE] =
```

The rule of index localization is true for reduction functions: domain declared as argument of reduction function has its own index system. The subject of this system action is arithmetical expression declared as the second argument of reduction function. It means that indexes used in the arithmetical expression may be divided into two types: "internal" which coincide with the indexes of the reduction domain and defined by them and "external" which don't coincide with indexes of reduction domain and defined by external domains (e.g. domains of other reduction functions or ASSUME operator). The set of "internal" indexes values is completely defined by the domain set as an argument of reduction function. The set of "external" indexes values is defined by the domains which are external for reduction function.

E.g. operator

```
FOR Oik ASSUME X=(i+k)/SUM( (Oij5) B.C.
```

defines request for computation

\[ X_{i,k} = \frac{i + k}{\sum_{i,m=1}^{n} (B_{i,j,m} \cdot C_{i,m})} \quad i, k = 1, \ldots, 10 \]

if the declarations are:

```
Oij5: ((i=1..5) ; (j=1..5)). Oik: (i=1..10) ; (k=1..10)).
Oijk: (Oijk((i=1..15) ; (j=1..15) ; (k=1..15))).
```
VARIABLE b DEFINED ON Oij. VARIABLE c DEFINED ON Oij.
VARIABLE x DEFINED ON Oik.

If external domain is conditional the condition acts on all the external indexes of arithmetical expression which are under the sign of reduction function. E.g. formula

\[ X_{i,j} = \sum_{i=1}^{100} B_{i,j} Y_j \quad i = 1,...,10, \quad j = 1,...,10, \quad i < j, \quad Y_i < 0, \]

sets computation

\[ Y_2 < 0: \quad X_{1,2} = \sum_{i=1}^{100} B_{i,2} Y_2 \quad \ldots \quad Y_{10} < 0: \quad X_{1,10} = \sum_{i=1}^{100} B_{i,10} Y_{10} \]

\[ Y_3 < 0: \quad X_{2,3} = \sum_{i=1}^{100} B_{i,3} Y_3 \quad \ldots \quad Y_{10} < 0: \quad X_{2,10} = \sum_{i=1}^{100} B_{i,10} Y_{10} \]

\[ \ldots \ldots \ldots \ldots \]

\[ Y_{10} < 0: \quad X_{9,10} = \sum_{i=1}^{100} B_{i,10} Y_{10} \]

may be written as:

Oij: Oi (i=1..10); Oj (i=1..10).
OILTj; Olj / i<1.
OILTjYLO, OILTjYGE0; OILTYj / Y<0.
O100: (i=1..100). Olij100: (O1000; Oj).

VARIABLE b DEFINED ON Oij100. VARIABLE x DEFINED ON Oij.
VARIABLE y DEFINED ON Oij.

FOR OILTjYLO ASSUME X=SUM ( O1000 B+y) .

Recursive computations are prohibited for the values used in both the left part of ASSUME operator and arithmetical expression in reduction function. E.g. formula

\[ X_{i,j} = \sum_{i=1}^{10} X_{i,j} \quad i = 2,...,10. \quad j = 1,...,5 \]

5.2.3.3 User's external functions.

External function is defined by user. One can call this function by name, the list of actual parameters divided by comma is put into brackets and placed after the name. The result of function's computations is represented by function's name. If all the actual parameters have their values computation is available. All the actual parameters are considered initial data for function computation. Side effect is prohibited.

Arithmetical expressions, part and function names or variables on domain (arrays) can be actual parameters of external function.

Here is an example of external function with different ways of actual parameters setting.
Grid: (Oi: (i=1..N); Oj: (j=1..M)).

VARIABLE A, B DEFINED ON Grid.
VARIABLE X, Y DEFINED ON Oi. VARIABLE T DEFINED ON Oj.
VARIABLE Gamma DOUBLE.

INPUT A, B ON Grid, Y ON Oi, T ON Oj. INPUT Gamma.

OUTPUT X ON Oi.

DOMAIN PARAMETERS N = 10, M = 20. EXTERNAL FUNCTION Dxy.

FOR Oi ASSUME X = f(Dxy, Gamma+0.5, Y) ON Oj, A ON Grid / (i=2..8, j=M-2), B ON Grid /j=1+2.

ASSUME operator set the rule of array \( X_{i,j} = i, 10 \) computation. User's function \( F \) with parameters values are to be computed for every \( i \). The list of parameters:

1-st parameter - name \( Dxy \) of external user's function;

2-nd parameter- arithmetical expression \( Gamma+0.5 \times Y_i \);

3-rd parameter - \( T_{j}, j=1, ..., 20 \) values of static array;

4-th parameter - \( A_{ij}, i=2, ..., 8, j=18 \) values of array's section: one-dimensional array consists of the 2-8 rows elements from the 18th column of \( A \) matrix;

5-th parameter - \( B_{ij}, i=1, ..., 10, j=i+2 \) of dynamic array's section: \( i+2 \) column of \( B \) matrix.

The rules of formal and actual parameters correspondence:

(1) The number of actual and formal parameters must be equal; correspondence is set from the left to the right in writing order.

(2)

<table>
<thead>
<tr>
<th>Formal parameter</th>
<th>Actual parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>function name</td>
<td>function name</td>
</tr>
<tr>
<td>part name</td>
<td>part name</td>
</tr>
<tr>
<td>scalar</td>
<td>arithmetical expression</td>
</tr>
<tr>
<td>variable on domain</td>
<td>variable on domain</td>
</tr>
</tbody>
</table>

(3) If both formal and actual parameters are variables on domain then the number of domain indexes and the number of points in the ranges on corresponding indexes must be the same. (5.1.2).

5.2.4 Part call.

call-part:

COMPUTE name-part [ (actual-parameters) ]
actual-parameters:

[list-in-parameter]

[RESULT list-out-parameter]

out-parameter:

name-scalar

name-variable-on-domain ON domain-of-parameter

iterated-variable-on-domain ON domain-of-parameter

Arithmetical expressions, part and function names or variables on domains (arrays) can be actual parameters.

Actual parameters may be declared as initial or results by key word RESULT.

E.g.:

COMPUTE Velocity ( Delta+0.5, Fi ON Oi,jk RESULT v ON Oi,j).

The first two parameters are initial, the third is result. Side effect is impossible: if sets of initial parameters and parameters-results intersect the result of intersection will be reassignment which is prohibited in NORMA.

Part call is a development of notion relation in ASSUME operator because it allows to obtain several results at once. Part call without ASSUME operator is a development of scalar operator notion.

The ways of part initial actual parameters setting and the ways of function actual parameters settings (5.2.3.3) are the same.

Scalars, variables with indexes (may be set by the rule of default indexes), variables on domains (arrays) may be actual parameters-results.

If part call is in the body of ASSUME operator then scalars and variables on static domain (which don’t change for different values of indexes from ASSUME header) can’t be parameters-values - it causes reassignment.

If part call isn’t in the body of ASSUME operator then scalar names, variables with index-constants and variables on static domains can be parameters-results only.

Here is an example of actual parameters settings in different ways.

VARIABLE A,B DEFINED ON Grid: ( Oi: (i=1..N);Oj: (j=1..M) ).

VARIABLE X,Y DEFINED ON Oi. VARIABLE Τ DEFINED ON Oj.

VARIABLE Gamma DOUBLE.

INPUT A ON Grid X ON Oi,T ON Oj. INPUT Gamma.

OUTPUT X ON Oi, B ON Grid /j=2..12.

DOMAIN PARAMETERS N =10, -20.

EXTERNAL PART DzDy.

FOR Oi ASSUME
COMPUTE TEST ( DzDy, T ON Oj, Y RESULT X, B ON Grid /j=1+2/).
COMPUTE SCALAR( A ON Grid RESULT Gamma, B ON Grid /j=2/).

The rule of $X_{ij}, i=1,\ldots,10$ and part of matrix $B : B_{ij}, i=1,\ldots,10, j=3,\ldots,12$ computation is defined in ASSUME operator.

Scalar call of part SCALAR defines scalar Gamma and the second column of B matrix.

The rules of actual and formal parameters-results correspondence are the same as for correspondence rules for functions(5.2.3):

<table>
<thead>
<tr>
<th>Formal parameter</th>
<th>Actual parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>scalar</td>
</tr>
<tr>
<td>scalar</td>
<td>variable with index constant</td>
</tr>
<tr>
<td>variable on domain</td>
<td>variable on domain</td>
</tr>
</tbody>
</table>

5.2.5 Interface with FORTRAN programs.

Subroutines and functions written in Fortran may be called from the program written in NORMA by usual tools of part and function call. Reassignment control is carried out in actual parameters analysis.

For example, subroutine

```fortran
SUBROUTINE SNIPPY(Y,N,X)
  REAL X(N), Y(N)
  DO 1 I=1,N
    X(I) = SIN(Y(I))
  1 CONTINUE
  RETURN
END
```

can be called as it is shown here:

O1 : (I=1,N).
VARIABLE X,Y DEFINED ON O1.
INPUT Y ON O1.
DOMAIN PARAMETERS N = 10.

COMPUTE SINXY ON O1, N RESULT X ON O1.

5.2.6 Setting of sequential computing mode.
The order of carrying out operations (operators) is often important in computational algorithms realization. The order may have an influence on convergence, stability of solution method etc. There is a possibility of sequential computations mode setting in Norma. It allows user to fix the order of computation. On this purpose delimiters `#` are used. Operators put in the delimiters `#` are performed in the order written in program. Correctness of operators' sequence in Norma is under control: reassignment, using of undeclared values, etc. are fixed. The notation

```plaintext
#
X=5.0 . Z=SIN(X+0.5) . Y=COS(X)-Z..X.
#
```

is correct, but notation

```plaintext
#
X=5.0. Y=COS(X)-Z..X . Z=SIN(X+0.5).
#
```

is incorrect because undefined value of Z variable is used in the second operator (if we take off delimiters then both notation are correct).

5.3 Iteration

**iteration:**

- **head-of-iteration**
  - [boundary-value]
  - initial-value
  - body-of-iteration
  - exit-condition
- **end-iteration**

**head-of-iteration:**

```
ITERATION list-iterated-element
ON name-iteration-index .
```

**iterated-element:**

```plaintext
name-variable{ (list-name-result) } 
```

**boundary-value:**

```
BOUNDARY { operator.} END BOUNDARY
```

**initial-value:**
/ INITIAL name-iteration-index =
  int-constant: {element-of-initial.}" END INITIAL/" element-of-initial :
  operator
declaration-of-input
declaration-of-output
body-of-iteration :
  {element-of-iteration-body.}" element-of-iteration-body :
  operator
  iteration
  declaration-of-output
exit-condition :
  EXIT WHEN log-expression
end-iteration :
  END ITERATION name-iteration-index

Computation process often is iterative in mathematical physics problems solutions. Such process may be set by previously described tools of the NORMA language. You should extend domains’ declarations adding extra direction corresponding to iteration index. But to use such an extension isn’t always correct because the direction is fictitious and it represents the way of computation but not space-time grid. Besides the boundaries of such fictitious direction are often unknown.

Special construction ITERATION allows possibility to set iterative computational process avoiding all the difficulties mentioned above.

Informally iteration set iterative computations with iteration index changing from 0 to some integer positive value which is a condition of iteration’s end.

Let’s take iterative computational process dealt with the system of equations solution:

\[ \sum_{j=1}^{m} A_{i,j} X_j = F_i, \quad i = 1, \ldots, m \]

and set by formulae

\[ X_{i+1} = \frac{1}{A_{i,i}} (F_i - \sum_{i \neq j} A_{i,j} X_j^i), \quad X_0^i = X0, \quad i = 1, \ldots, m \]

exit condition \[ \| X^i - X^{i-1} \| < \varepsilon \]

This process written in NORMA:
ARRAY: (Oj: (I=1..M); Oj: (J=1..M)).

VARIABLE X0,X,P DEFINED ON OI. VARIABLE A DEFINED ON ARRAY.

VARIABLE Epsilon.

DOMAIN PARAMETERS M = 100.

INPUT X0 ON OI, A ON ARRAY. INPUT Epsilon.

OUTPUT X, Xpred ON OI.

ITERATION X (Xpred) ON N.

INITIAL N=0 :

FOR OI ASSUME X = X0.

END INITIAL

FOR OI ASSUME

X = 1/A[j=1..M]SUM( (Oj[i<!>)A.X[I=1..N,J,N-1]).

EXIT WHEN MAX( (OI) ABS(X[N] - X[N-1]) ) < Epsilon

END ITERATION N.

Iteration itself is the last 8 lines of example. There is iteration index (here it is N) and variables taking part in computations and presenting the result of these computations (- X -is the value from the last iteration step, - and Xpred — the value from the before the last iteration step) in the iteration header. Value Xpred doesn’t require supplementary definition, it is considered to be declared in the same way as X. Values X and Xpred can be used for computations out of iteration: in the given example these values are declared output.

You can use the possibility of iterated variables boundary values of setting (in general it is not obliged). Boundary values of iterated variable are set by common Norma operators inside the block BOUNDARY ... END BOUNDARY. These values are considered unchangeable on iteration and defined on every its step.

Initial values for iterated variables are set by blocks INITIAL...END INITIAL (there may be several ones). If the values of iterated variables depend only on the values of the previous level (one-level iteration) then only one block is indicated

INITIAL N=0:... END INITIAL.

If the values of iterated variable depend on the values of two previous levels (two-level iteration) then two blocks are indicated

INITIAL N=0: ... END INITIAL

INITIAL N=1: ... END INITIAL

etc.

Variables which are the result of multilevel iteration and used outside iteration are indicated in the header of iteration. E.g. the header

ITERATION X(X1,X2,X3) ON N.
defines values X,X1,X2,X3 of iterated value X obtained at the last four steps of four level iteration. The number of results can't be more than n for n-level iteration.

Body of iteration is part of NORMA program. In particular you can define new iteration in another direction inside the iteration. In the given example body of iteration consists of the only ASSUME operator.

Iteration index can be used in the list of indexes indicated for iterated variable. In fact we can consider that iterated variable has supplementary index in boundaries of iteration (on factitious direction). Iteration index without displacement may be not indicated.

Only iterated variables can indicate iteration index.

The values of variables unindicated in the list of iterated variables may be computed in the body of ITERATION construction. Such variables may be used for interim results of each level iteration representation. The usage of such variables doesn't break reassignment prohibition: we consider the a new copy of the variable is used at each step of iteration.

Iteration process ends if logical expression set in exit condition becomes true.
Appendix 1. Representation of initial program.

Initial program is represented in initial file according to the rules:

1. Text of every part of NORMA program is written in unformatted representation. If you need to carry operator or declaration it is prohibited to break key words, identificators, constants. In other cases you can carry the words as you like it.

2. Key words, identificators, constants are separated by \textit{spaces, special symbols, end-line symbol}.

   Spaces are not meaningful symbols: the group of spaces are considered as one space.

3. The line or part of line beginning from symbol "!" is a \textit{commentary}.

4. Information placed in the interval from the symbol "?" standing at the first position up to the symbol "?" also standing at the first position are not translated. It allows to choose from initial file those parts which you want to translate.
Bibliography


Keldysh Institute of Applied Mathematics
Academy of Science USSR

Andrianov A.N.

Organization of loop computations in the NORMA language.

Moscow 1986
The problems of loop process organization for the program written in nonprocedural language NORMA are considered in this paper. An algorithm of designing the system of simple loops allowing parallel processing is given. The algorithm is based on the notion of computation's front which is a hyperplane where variables' values may be computed in every its point. The task of Linear Integer Programming is solved for determination of the hyperplane's parameters.

Key words: nonprocedural language, synthesis of the program, parallel computations.
Contents.

1. Introduction 3
2. Principal NORMA operator 3
3. Determination of the operator's order 4
4. Front of computations 5
5. Statement of Linear Integer Programming problem 8
6. Program scheme for the one operator 9
   7. Structure for index expressions 14
8. Front of computations with complicated parameters 15
9. Loop structure for some relations in the case of simple front 17
10. Examples 19
11. Bibliography 23
1. Introduction.

The NORMA language is used for writing numerical solutions of mathematical physics problems. Program in NORMA is a strict notation of formulae and some supplementary information for a translator to design computer program.

Program has no information about the order of computation and the ways of loop process organization. Determination of computation order is one of the main translator's tasks. Information connections set in the program are initial data for computation order's determination and organization of loop process.

Nonprocedural nature of NORMA allows to overcome some difficulties in algorithm parallelization.

An algorithm of loop process organization for NORMA program is given in this paper. Loop parallel processing can be organised by this algorithm.

Principal NORMA operator.

There are declarative and executive parts in NORMA program. Variables names types which are to calculate in the task's computation are indicated in declarations. The rules of carrying out computations according to the language semantics are placed in the executive part [2].

Let's describe briefly principal language operator ASSUME. The structure of the operator is given below:

FOR <domain> ASSUME <relation>.

Let's determine informally the terms "domain" and "relation". The domain is a set of points with integer non-negative coordinates in n-dimensional space.

The form and the type of relation is given below:

\[ X_{i_1, i_2, ..., i_n} = f(X_{A_1}, ..., X_{A_m}), \quad i = i_1, i_2, ..., i_m \]

Here \( X \) are variables names. Set \( \{i_1, i_2, ..., i_n\} \) is a set of indexes. Index construction \( A_k \quad k = i_1, i_2, ..., i_m \) has the following structure:

\[ i_{k_\mu} \pm \Delta_{k_\mu}, \ldots, i_{k_\mu} \pm \Delta_{k_n} \quad 1 \leq \mu \leq n \]

\( \Delta_{k_\mu} \) - integer non-negative constants.

Relation sets the rule of deriving \( X \) value from the left part of the relation by the values of variables from the right part. Any variable is calculated only once in every point of domain i.e. reassignment is prohibited in NORMA. Operator ASSUME points out the necessity of computing the variable's values from the left part of the relation for all the points of the domain set in the operator's
header. In general it doesn't mean that computing of $X$ will be carrying out simultaneously for all the points of the domain in the particular place of the program.

Example. Let two operators be set:

1. FOR $A$: $(k=1..m)$ ASSUME  
   $X=f_1(Y_{k+1})$.

2. FOR $A$ ASSUME  
   $Y=f_2(X_k)$

We consider value of variable $Y$ for $K=0$ has been calculated before these operators. It is evident that operators order where all the values of $X$ variable are calculated first and $Y$ values are done after them is impossible. Argument is to be computed for calculating $X$ even when $K=2$ (here argument is $Y$ value for $K=1$). $Y$ is calculated in the second operator i.e. $X$ and $Y$ components are computed by turn in the given case.

Determination of the operators' order.

One of the problems to be solved on the translation stage is the problem of operators (or the groups of the operators) order determination.

To specify a program as a dependence graph is a traditional way of program representation and we use well-known approaches and methods in our work.

Let's determine dependence graph $G(V, E, \lambda)$ which is a directed graph. $V$ denotes a set of nodes corresponding to the names taking part in the computations. If variable $X$ is calculated in several operators it corresponds to several nodes. $E$ denotes a set of arcs corresponding to relations between variables. Arc $(\Delta_{k_1}, \ldots, \Delta_{k_n})$ from the node corresponding to variable $X$ to the node corresponding to variable $X$ conforms to the dependence $X$ on the values of $X$ with index structure $i_k \pm \Delta_{k_1}, \ldots, i_k \pm \Delta_{k_n}$. $\lambda$ denotes a set of arcs' marks.

Strongly connected subgraph of graph $G$ not being included into any other strongly connected subgraph of graph $G$ is called the Most Strongly Connected Subgraph (MSCG) of graph $G$. Having chosen all the MSCG $G_1, \ldots, G_n$ (e.g. by the method suggested in [3]) partial arrangement of $G_1, \ldots, G_n$ could be done: if there is an arc from $G_i$ to $G_j$, then $i > j$.

Consider loop program realization $R(G)$ built for every $G_i$. Then the program on the set graph $G$ can be represented in the form of realizations' $R(G_i)$ sequence taking into
consideration partial arrangement set earlier. Thus let’s consider an algorithm of designing loop realization for the operators corresponding to the nodes of MSCG.

Loop program-realization is designed for particular operators. Further we’ll determine the class of these operators.

Consider relation system for computing values \( X_1, \ldots, X \):

\[
X_{i_k, b, \ldots, b} = f_1(X_{k \notin A_{1}^{1}, \ldots, k \notin A_{1}^{m}}, \ldots, X_{k \notin A_{m}^{1}, \ldots, k \notin A_{m}^{m}})
\]

\[
X_{i_k, b, \ldots, b} = f_m(X_{k \notin A_{1}^{1}, \ldots, k \notin A_{1}^{m}}, \ldots, X_{k \notin A_{m}^{1}, \ldots, k \notin A_{m}^{m}})
\]

(1)

Initial domain is set by inequalities:

\[ M_k \leq i_k \leq N_k \quad k = 1, \ldots, n \]

(2)

System (1) is said to correspond to some MSCG.

Task may be formulated in the following way:

Design loop program-realization which provides computation of variables \( X_1, \ldots, X \) for all the points of the domain (2) if such computation is possible. Program realization must be designed according to the given system of relations (1).

**Front of computations.**

Further we’ll consider computation organization only in the formulae corresponding to MSCG. Considered model of computations is realized in several stages. The first stage consists of computing the variables depended on external data (i.e. the values of such variables is computed outside given MSCG). The set of points where variable \( U \) is calculated at the first stage is indicated as \( S_{U}^{1} \). The second stage consists of computing the variables depended on both external data and the variables computed at the first stage. The set of points where the variable \( U \) is calculated at the second stage is indicated as \( S_{U}^{2} \). This process will be continued till \( U \) values are computed in all the points of the initial domain. The sequence of \( U \) variable computation domains \( S_{U}^{1}, \ldots, S_{U}^{n} \) will be the result of the process. Computation of \( U \) variable in all the points of \( S_{U} \) is carried out parallel but transition from the one domain to another is sequential. Similar domains systems are to be designed for all the variables included into MSCG.
Complicity of the sets \( S_u \) structures caused impossibility of writing the traversal of all
the points from \( S_u \) domains in the form of effectively realized loops' systems.

On the other hand the following representation seemed acceptable: a set of points belonged to
the domain can be set by an equation of hyperplane given below:
\[
L_u(k): \ p_1^i i_1 + \ldots + p_n^i i_n + \Delta_n = k
\]  
where \( p_1^i, \ldots, p_n^i, k \) - integers.

We arrange sets \( L_u(j) \) in the following way. Let \( L_u(1) \) contains the points where the
computation of \( U \) values depends on the external data only and these points satisfy a condition given below:
\[
\ p_1^i i_1 + \ldots + p_n^i i_n + \Delta_n = 1
\]
Set \( L_u(2) \) consists of the points where the computation of \( U \) values depends on the external data and
on the values computed in the points of set \( L_u(1) \) besides these points satisfy a condition given below:
\[
\ p_1^i i_1 + \ldots + p_n^i i_n + \Delta_n = 2
\]
We'll continue this process till all the values of variable \( U \) in the initial domain are computed
Example: Let’s consider relations:
\[
X_{i, j} = f_1(X_{i-1, j+1}; Y_{i-1, j+2})
\]
\[
Y_{i, j} = f_2(X_{i+2, j-1}; Y_{i-3, j+1})
\]
set on the domain \( 1 \leq i \leq 7, \ 1 \leq j \leq 7 \)

The value in the \((i,j)\)square points out number \( K \) of corresponding set \( S^k \). E.g. the value in
\((4,4)\) square of \( X \) variable equal to 7 means that variable \( X \) in the point \((4,)\) is computed on the 7th step of model's functioning.
Difficulties in the designing of loops’ system for effective traversal of $S^k$ set points are evident.

Let’s create the system of sets belonging to the plane that is defined by the equations:

\[ L_x(k): \quad -i + 2j + 4 = k \]
\[ L_y(k): \quad -i + 2j + 5 = k \]

The tables where the values in the squares determine belonging of a particular point to the set are given below:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & \hline
10 & 9 & 8 & 7 & 6 & 5 & 4 & \hline
12 & 11 & 10 & 9 & 8 & 7 & 6 & \hline
14 & 13 & 12 & 11 & 10 & 9 & 8 & \hline
16 & 15 & 14 & 13 & 12 & 11 & 10 & \hline
18 & 17 & 16 & 15 & 14 & 13 & 12 & \hline
20 & 19 & 18 & 17 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1
\end{array} \]

Differed from sets $S^k$ sets $L(k)$ have regular structure. It gives a chance to write the sequence of computations in the form of not complicated loop structure.

Given tables show us that at each step of computation there is a set of points where variables values are computed simultaneously (parallel).

E.g. the values of variable $Y$ in the points (1, 7) (3, 6) (5, 5) (7, 4) and the values of variable $X$ in the points (2, 6), (4, 5), (6, 7) are computed parallel at step $K=8$.

All the points where variable $X$ is computed parallel lie on the one line.

The values of $Y$ in the points (2, 6) (4, 5) (6, 4) and the values of $X$ in the points (1, 6) (3, 5) (5, 4) (7, 3) are computed at the next step. All the points where variable $Y$ is computed at this step also lie on the one line.

Let’s assume that there is such a hyperplane where we can organize computations in the way described above. Introduce a definition.

**Definition.** Front of computation of variable $U$ at the step $K$ is a set of points \( \{i_1, \ldots, i_n\} \) where the value of $U$ may be computed parallel and the points belong to hyperplane $L_u(k)$: $\rho^1i_1 + \ldots + \rho^n_i + \Delta_n = k$.
Changing $K$ values we obtain the family of hyperplanes which covers all the points of the initial domain.

We can show that fronts of computations included into MSCG and satisfy condition $i_k \geq 0 \quad k=1,n$ are parallel hyperplanes.

Consider the relation included into MSCG

\[ U_{i_1, i_2, \ldots, i_n} = f_u(i_1; i_2; \ldots; i_n + a_n + \ldots) \]

point $(i_1, \ldots, i_n) \in L_u(k_0)$, point $(i_1 + a_1, \ldots, i_n + a_n) \in L_v(k_i)$

the value of $V$ must be computed before $U$ value is available, so

\[ \rho_i + \rho_n + \Delta_n = k_0 > k_i = \rho_i + \rho_n + \Delta_n + \Delta_v \]

The main inequality is derived from the one given above. All the parameters of hyperplane must satisfy it

\[ \Delta_u - \Delta_v - \sum_{k=1}^{n} \rho_k a_k > 0 \quad (5) \]

Consider computations' fronts relations for variables $U$ and $V$ included into MSCG.

\[ L_u(k): \sum_i \rho_i + \Delta_u = k \]

\[ L_v(k): \sum_i \rho_i + \Delta_v = k \]

Let's consider some point with coordinates $(i_1^0, \ldots, i_n^0)$ belonged to the initial domain. The computation of $U$ value in this point is carried out at $K_u$ step and the computation of $V$ value is done at $K_v$ step, i.e.

\[ \sum_i \rho_i^0 + \Delta_u = K_u \]

\[ \sum_i \rho_i^0 + \Delta_v = K_v \]

Subtracting the second equation from the first one we obtain:

\[ \Delta_u - \Delta_v = K_u - K_v \]

i.e. difference $\Delta_u - \Delta_v$ shows us "delay" ("outstripping") of $U$ variable in comparison with $V$ variable.

Now we can divide the question of loop process organization into two questions.

1. Definition of computation's front hyperplane's parameters.

2. Definition of coordinates of the points lying on the front's hyperplane (i.e. determination of the parameters of the loops providing traversal of all the hyperplane's points).

Statement of Linear Integer Programming problem.
Let the system of relations (1) be set. Parameters \((p_1, \ldots, p_n)\) are to determine for variables \(X, \ldots, X\). Besides values of \(\Delta_i\) are to be defined for every variable \(X\). System of inequality (5) defines the conditions for \(p_1, \ldots, p_n, \Delta: \quad i = 1, m\) to satisfy.

Computations in the points lying on the one hyperplane are done parallel. But transition from the one plane to another is sequential. Thus the number of the planes covering all the points of the initial domain determines \(K_{\rho}\) number of a sequential steps of computations.

The formula for \(K_{\rho}\) is given below.

\[
K_{\rho} = \sum_{i=1}^{n} |p_i|^* (N_i - M_i)
\]

where \(N_i, M_i\) are the upper and the low boundaries of changing \(i\) coordinate domain.

Thus the task of wave’s front determination comes to solution of LIP problem of the type given below.

Find \(p_1, \ldots, p_n, \quad \Delta_1, \ldots, \Delta_m\) which satisfy

\[
\min \sum_{i=1}^{n} |p_i|^* (N_i - M_i)
\]

with following constraints

\[
\Delta_i - \Delta_i - \sum_{k=1}^{n} p_k \delta_{ii} > 0 \quad i, l = 1, m
\]

where \(\delta_{ii}\) is a displacement to index \(i\) in the relation defined dependence of \(X\) on \(X\) during the computation.

Nowadays direct algorithm of integer programming is used for solving such a task [4].

Note 1. Further we’ll consider only those MSCG where formulated task of LIP has a solution.

We don’t consider the question of computation organization in the case of sequential computing. E.g. for the relations of the following type

\[
A_{i,j} = f_1(A_{i-1,j+1}; B_{i,j})
\]

\[
B_{i,j} = f_2(A_{i-1,j}; B_{i-1,j+1})
\]

with \(2 \leq i \leq M, 2 \leq j \leq N\)

there is no front of computations (as it is defined above). It is derived from the fact that inequalities of (5) type for the given example has the following form

\[
p_1 - p_2 > 0
\]

\[
-\Delta_n > 0
\]
\[ \Delta u + p_2 > 0 \]
\[ -p_1 + p_2 > 0 \]

We can see from the given inequalities (the first and the last inequalities are conflicting) that there is no values \( p_1, p_2 \) defining the front of computations for variables \( A \) and \( B \) but the computations for the given relations may be organized in the form of the simple sequential loops.

Note 2. Presence of absolute values in the function demands solving of several tasks and choosing of the optimal one. We have to face the real tasks with the dimension of no more than 3 and it allows to judge of practical applications of this approach.

Program scheme for the one operator.

Let's assume that variable's values computed in the left part are not used in the right part i.e. there is no recurrent dependence. It means that all the values of the used variables are known and the computation can be done parallel for all the points of the domain. The scheme of operators' loop has the following form:

\[
\begin{align*}
\text{DO LABEL } & i_1=M_1,N_1 \\
& \ldots \ldots \\
\text{DO LABEL } & i_n=M_n,N_n \\
\text{LABEL } & x(i_1,i_2,\ldots,i_n)=\ldots\ldots\ldots\ldots
\end{align*}
\]

Now assume that there is a recurrent dependence in the operator. In general the task of finding the parameters of wave's front is to be solved first and then you may start designing the loops' scheme. The exception might be such an index (or the group of indexes) which displacements has the one sight. If such an index exists we won't consider it. The header of the loop corresponding to particular index has the form:

\[
\begin{align*}
\text{DO LABEL } & i_k=M_k,N_k \\
\text{DO LABEL } & i_k=M_k,N_k-1
\end{align*}
\]
in the case of the negative displacements.

in the case of the positive displacements.

Let's create a new graph \( G' \) derived from the initial graph \( G \) by deleting the arcs where corresponding mark is strictly positive. Then use procedure of finding index with displacements of one sign. If there is no one like that then for the rest dependencies on other indexes you'll have to solve the task of determination of computations' front parameters.

Hyperplane of computations' front is said to be defined. The scheme of the program in this case will have the form:

\[
\begin{align*}
\text{DO LABEL } & K=K_b,K_e \\
\text{DO LABEL } & \text{CONC ALL } (i_1,\ldots,i_n) \in L(k)
\end{align*}
\]
where \( L(k) \) is an equation of hyperplane. Record CONC ALL \((i_1, \ldots, i_n) \in L(k)\) means that the loop provides processing of all the points lying on the hyperplane \( L(k) \) and the computation in this points may be carried out parallel. External loop is sequential and it provides transition from the points of the one hyperplane to another.

Front of computations has the form:

\[
p_{11}i_1 + \ldots + p_{m1}i_n = k
\]

(In the case of one relation \( \Delta = 0 \))

Changing \( K \) values we obtain the system of hyperplanes which covers all the points lying in the initial domain. It is evident that minimal and maximal values of \( K \) variable are on the domain's boundaries and they are equal to:

\[
K^\prime_b = \sum_{p_i < 0} p_i N_i + \sum_{p_i > 0} p_i M_i \quad (6)
\]

\[
K^\prime_s = \sum_{p_i < 0} p_i M_i + \sum_{p_i > 0} p_i N_i \quad (7)
\]

Now let's consider the question of designing loops' system providing processing of all the points of hyperplane. First let's assume that all \( p_i \neq 0 \quad i = 1, \ldots, n \) and the greatest common divisor of the numbers \( p_i \) are equal to 1.

Indicate \( t_i = p_{i1} i_1 + \ldots + p_{in} i_n \quad (8) \)

Initial equation assumes the form

\[
p_{11}i_1 + t_i = k - \Delta \quad (9)
\]

Solution of this equation depends on the parameter \( d_i \) (integer) and has the form:

\[
i_i = d_i
\]

\[
t_i = (k - \Delta) - p_i d_i \quad (10)
\]

It is natural that you need to choose only those solutions which belong to the initial domain. E.g. value \( i_i \) must satisfy inequalities

\[
M_i \leq i_i \leq N_i
\]

i.e. \( M_i \leq d_i \leq N_i \). Besides let's define the boundaries of changing value of \( t_i \).

Indicate

\[
t_i^{\text{max}} = \sum_{p_i < 0} p_i N_i + \sum_{p_i > 0} p_i M_i \quad (11)
\]
\[ t_{\text{min}}^1 = \sum_{i=2}^{n} p_i N_i + \sum_{i=2}^{n} p_i M_i \quad (12) \]

It is evident that \( t_{\text{min}}^1 \leq t_1 \leq t_{\text{max}}^1 \). Substituting instead of \( t_1 \) its expression into the last inequalities and solving it on \( d_1 \) we obtain

\[
\frac{t_1^0 - t_{\text{min}}^1}{p_i} \geq d_1 \geq \frac{t_1^0 - t_{\text{max}}^1}{p_i} \quad \text{if} \quad p_i > 0
\]

and

\[
\frac{t_1^0 - t_{\text{max}}^1}{p_i} \geq d_1 \geq \frac{t_1^0 - t_{\text{min}}^1}{p_i} \quad \text{if} \quad p_i < 0
\]

where \( t_i^0 = k - \Delta \)

In the result the domain of changing value of \( d_1 \) is set by inequalities

\[
d_i^0 \geq d_1 \geq d_i^b
\]

where

\[
d_i^b = \left[ \max(M_1, \frac{t_1^0 - t_{\text{max}}^1}{p_i}) \right]
\]

\[
d_i^c = \left[ \min(N_1, \frac{t_1^0 - t_{\text{min}}^1}{p_i}) \right]
\]

(13)

for \( p_i > 0 \)

\[
d_i^b = \left[ \max(M_1, \frac{t_1^0 - t_{\text{max}}^1}{p_i}) \right]
\]

\[
d_i^c = \left[ \min(N_1, \frac{t_1^0 - t_{\text{min}}^1}{p_i}) \right]
\]

(14)

for \( p_i < 0 \)

Consider the rest part of the formula

\[ p_{i_2} + \ldots + p_{i_n} = t_2^0 \]

where \( t_2^0 = (k - \Delta) - \mu d_1 = t_1^0 - \mu d_1 \)

Indicate

\[ t_2 = p_{i_2} i_2 + \ldots + p_{i_n} i_n \]

Equation (15) assume the form

\[ p_{i_2} i_2 + t_2 = t_2^0 \]

Its solution has the form

\[ i_2 = c_2 \]
\[ t_2 = t_2^0 - p_2 d_2 \]

Let's define
\[
\begin{align*}
t_{\text{max}}^2 &= \sum_{i=3}^{n} p_i N_i + \sum_{i=3}^{n} p_i M_i \\
\quad &\quad p_i > 0 \\
\quad &\quad p_i < 0
\end{align*}
\]
\[
\begin{align*}
t_{\text{min}}^2 &= \sum_{i=3}^{n} p_i N_i + \sum_{i=3}^{n} p_i M_i \\
\quad &\quad p_i < 0 \\
\quad &\quad p_i > 0
\end{align*}
\]
From the inequalities
\[ M_2 \leq i_2 \leq N_2 \]
\[ t_{\text{max}}^2 \geq t_2 \geq t_{\text{min}}^2 \]
we obtain the range of changing the value
\[ d^2 \geq d_2 \geq d^2 \]
where
\[
d^2 = \left[ \frac{\max(M_2, t_2^0 - t_{\text{max}}^2)}{p_2} \right]
\]
\[
d^2 = \left[ \frac{\min(N_2, t_2^0 - t_{\text{min}}^2)}{p_2} \right]
\] (16)
for \( p_2 > 0 \). In the case of \( p_2 < 0 \) the formulae are derived in the same way.
We'll continue this process till the initial equation assumes the form
\[ p_{n-1} i_{n-1} + p_n i_n = t_{n-1}^0 \] (17)
Let's consider that \( p_{n-1} \) and \( p_n \) are reciprocals. Then at first using Euclid's algorithm, we solve the equation
\[ p_{n-1} i_{n-1} + p_n i_n = 1 \] (18)
Let \( \hat{i}_{n-1}, \ldots, \hat{i}_1 \) be the solution of the equation (18). Then it will be evident that
\[
i_{n-1} = \hat{i}_{n-1} t_{n-1}^0 + d_{n-1} \cdot p_n
\]
\[
i_n = \hat{i}_n t_{n-1}^0 + d_{n-1} \cdot p_{n-1}
\]
are the solution of equation (17). From the conditions
\[ M_{n-1} \leq i_{n-1} \leq N_{n-1} \]
\[ M_n \leq i_n \leq N_n \]
we obtain the range of changing \( d_{n-1} \) value
\[ d_{n-1}^e \geq d_{n-1} \geq d_{n-1} \]
where \( \alpha_{n-1}^o = \left[ \max \left( \frac{M_{n-1} - t_{\alpha}}{p_n}, \frac{t_{\beta} - N_n}{p_{n-1}} \right) \right] \) (19)

\( \alpha_{n-1}^e = \left[ \min \left( \frac{N_{n-1} - t_{\alpha}}{p_n}, \frac{t_{\beta} - M_n}{p_{n-1}} \right) \right] \) (20)

in the case of \( p_{n-1}, p_n > 0 \)

where \( t_{\alpha} = \alpha_{n-1}^o \times t_{n-1}^o \quad t_{\beta} = \alpha_{n-1}^e \times t_{n-1}^e \)

The rest expressions for \( \alpha_{n-1}^o, \alpha_{n-1}^e \) values with the different combinations of the signs \( p_{n-1}, p_n \) values are derived in the same way. Thus the scheme of the loop provided processing of all the points of hyperplane has the following form:

\[ t_i^0 = k - \Delta \]

\[ D_i^o = \ldots \]

\[ D_i^e = \ldots \]

DO LABEL \( D_i = D_i^o, D_i^e \)

\[ t_2^0 = t_1^0 - p_{\alpha} \times D1 \]

\[ D_2^o = \ldots \]

\[ D_2^e = \ldots \]

DO LABEL \( D_2 = D_2^o, D_2^e \)

\[ t_{\alpha} = \alpha_{n-1}^o \times t_{n-1}^o \]

\[ t_{\beta} = \alpha_{n-1}^e \times t_{n-1}^e \]

\[ D_{n-1}^o = \ldots \]

\[ D_{n-1}^e = \ldots \]

DO LABEL \( D_{n-1} = D_{n-1}^o, D_{n-1}^e \)

LABEL

\[ \quad \]

Note: Assume that as some index \( i_k \) exists then parameter of hyperplane \( p_k = 0 \). Hence the equation has the form:

\[ p_1 i_1 + \ldots + p_{n-1} i_{n-1} + p_{n-1} i_{n-1} + \ldots + p_n i_n + \Delta = k \] (21)

In this case we consider algorithm of loop designing for the equation as it is given above (21). The header of the loop corresponding to the direction \( i_k \) will have a form:

DO LABEL \( i_k = M_k, N_k \)
Loop on the direction $i_k$ can be formulated as both external and embedded.

Structure of index expressions.

The order of the embedded loops for processing of all the points of hyperplane are determined above. Each direction of the space corresponds to some loop index. For the first $n-2$ directions correspondence has the form:

$$i_k = d_k$$

For the last two indexes correspondence has more complicated form:

$$i_{n-1} = t_\alpha + d_{n-1} \cdot p_n$$  
$$i_n = t_\beta + d_{n-1} \cdot p_{n-1}$$

I.e. index structure of the relation $(i_1, ..., i_r)$ corresponds to the following structure in the program:

$$(d_1, ..., d_{n-2}, t_\alpha + d_{n-1} \cdot p_n, t_\beta - d_{n-1} \cdot p_{n-1})$$

The numbers $p_{n-1}, p_n$ must be reciprocals for the solution of equation(18). Let them be non-reciprocals. Then before start an algorithm we must find two reciprocals among $n$ numbers of $p_1, ..., p_n$. Such numbers must be found by all means because at the beginning we assume that the greatest common divisor of the numbers $p_1, ..., p_n$ are 1. This constraint doesn't change an algorithm.

In this case only the correspondence of the space directions and loop indexes is changed. The correspondence for this case is given below.

$$i_1 = d_1$$

...............

$$i_{r-1} = d_{r-1}$$

$$i_{r+1} = d_r$$

...............

$$i_{s-1} = d_{s-2}$$

$$i_s = d_{s-1}$$

.............

$$i_n = d_{n-2}$$

$$i_r = t_\alpha + d_{n-1} \cdot p_s$$

$$i_s = t_\beta + d_{n-1} \cdot p_l$$

Front of computations with complicated parameters.
Previous computations were carried out having assumed that the greatest common divisor \( (\delta) \) differs from 1. Let's consider changes in the algorithm in the case \( \delta \neq 1 \).

Let the initial front has the form:
\[
L_1(k): \quad \delta(p_i + \ldots + p_n) + \Delta = k
\]

Represent \( \Delta \) in the form of \( \Delta = \delta \bar{\Delta} + r, \quad 0 \leq r \leq \delta \)

Let's show that there are the points on the hyperplane for every value of \( K \). Let the point with coordinates \( (p_1^i, \ldots, p_n^i) \) lying on the hyperplane \( L_1(k_0) \) be at the step \( k_0 \). Assume that the point on the hyperplane \( L_1(k_0 + t) \) exists at the step \( k_0 + t, \quad t < \delta \). Then
\[
\delta(p_i + \ldots + p_n) + \Delta = k
\]
\[
\delta(p_i + \ldots + p_n) + \Delta = k_0 + t
\]
are the equations where the following formula is derived from
\[
p_i(-i_1^i - i_2^i) + \ldots + p_n(-i_n^i - i_2^i) = \frac{t}{\delta} \quad (23)
\]

There is an integer value in the equation from the left but there is none from the right. Hence the statement of the point existence on the hyperplane \( L_1(k_0 + t) \) isn't right. Further more from the last equation we can see that for the initial point of the hyperplane lie on the planes which are \( \delta \) value apart, i.e. if there are the points on the plane \( L_1(k_0) \), then the new points lie on the planes \( L_1(k_0 + l\delta) \) where \( l = \pm 1, \pm 2 \ldots \).

**Proposition.** The sequence of the computations for the front
\[
L_2(k): \quad p_i + \ldots + p_n + \bar{\Delta} = k \quad (24)
\]
is equivalent to the sequence of the computations for the initial front \( (22) \).

**Proof.** First let's show that at the initial value \( K = K^1_0 \) for the initial front the computations are carried out in the same points and for the front in the form \( (24) \) for \( K = K^2_0 \) where
\[
K^2_0 = \sum_{\rho > 0} M_\rho + \sum_{\rho < 0} N_\rho + \bar{\Delta}
\]
and \( \bar{\Delta} \) is taken from the representation \( \Delta = \delta \bar{\Delta} + r \)

From the representation \( K^1_0 \) and \( K^2_0 \) there \( K^2_0 = K^2_0 + \delta + r \) is derived

Consider point \( (p_1^i, \ldots, p_n^i) \in L_1(K^2_0) \). Let's show that this point belongs to the front of \( (24) \)
type for \( K = K^2_0 \) i.e. \( (p_1^i, \ldots, p_n^i) \in L_2(K^2_0) \); \( (p_1^i, \ldots, p_n^i) \in L_1(K^1_0) \), i.e.
\[
\delta(p_i + \ldots + p_n) + \Delta = K^1_0 \quad (25)
\]
We must show that
\[
p_i^0 + \ldots + p_n^0 + \bar{\Delta} = K^2_0
\]
Substitute expression $K_b^i$ into (25)

$$\delta (p_1^0 + ... + p_n^0) + \Delta = K_b^2 * \delta + r$$

Taking into consideration that $\Delta = \delta \Delta + r$ we have

$$\delta (p_1^0 + ... + p_n^0 + \Delta) + r = K_b^2 * \delta$$

i.e.  

$$p_1^0 + ... + p_n^0 + \Delta = K_b^2$$

We have shown that the points lying on the front's plane exist not for every $K$ for the initial equation.

If the first are on plane $L_1(K_b)$ then the other lie on plane $L_2(K_e + \delta * m)$. Let's show that for any point lying on this plane there exists the plane of front set in the form (24)

$$L_2(K_e^2 + m)$$

Let $(\bar{p}_1, ..., \bar{p}_n) \in L_1(K_b^1 + \delta * m)$

Let's show that given point

$$(\bar{p}_1, ..., \bar{p}_n) \in L_2(K_b^1 + m)$$

$$\delta (p_1^0 + ... + p_n^0 + \Delta) + r = K_b^1 + \delta m$$

We must show $p_1^0 + ... + p_n^0 + \Delta = K_b^2 + m$

We have $K_b^1 = K_b^2 * \delta + r$

Substituting the expression for $K_b^1$ we obtain

$$\delta (p_1^0 + ... + p_n^0 + \Delta) + r = K_b^2 * \delta + r + \delta * m$$

$$p_1^0 + ... + p_n^0 + \Delta = K_b^2 + m$$

It is derived from the proposition 1 that in the case $\delta \neq 1$ there is enough to transit from the front of (22) type to the front of (24) type and then to use the algorithm of loop parameters definition given above.

Loop structure for some relations in the case of the simple front.

Let the front for $X$ variables included into MSCG has the form

$$L_j(k): p_1^0 + ... + p_n^0 + \Delta_j = k \quad j = \overline{1, m}$$

The dimension of all the variables is said to be $n$ and $p_1, ..., p_n$ are reciprocals. As appears from the above if the dimensions of the domains and $\Delta$ values are the same for the variables then loop headers and the same too. Thus in this case the formulae setting computations for such variables may be written together in the body of designed loops' system. If this condition isn't performed then first we define $K_e$ and $K_e$ as

$$K_e = \min_i K_i^1, \quad K_e = \max_i K_i^1 \quad j = \overline{1, m}$$
Further we must put the test of the following type before every structure

$$K_{b} \leq K \leq K_{e}$$

and only when the condition is performed we'll start the loop providing search of the points on the front's plane with current $K$ value for the given variable.

**Note.** Though if there are still variables set on the one domain and had the same $\Delta$ value they may be united.

Thus the scheme of loop in this case has the following form:

$$K_{b} = \ldots$$

$$K_{e} = \ldots$$

DO 

**LABEL** $K = K_{b} \cdot K_{e}$

IF 

$$K \geq K_{b}^{l} \land K \leq K_{e}^{l}$$

DO 

**LABEL** $j$ CONC ALL $(i_{1}, \ldots, i_{n}) \in L_{j}(K)$

$$x(i_{1}, \ldots, i_{n}) = \ldots$$

............................................

**LABEL** $j$

$$x(i_{1}, \ldots, i_{n}) = \ldots$$

............................................

IF 

$$K \geq K_{b}^{r} \land K \leq K_{e}^{r}$$

DO 

**LABEL** $r$ CONC ALL $(i_{1}, \ldots, i_{n}) \in L_{r}(K)$

$$x(i_{1}, \ldots, i_{n}) = \ldots$$

............................................

**LABEL** $r$

$$x(i_{1}, \ldots, i_{n}) = \ldots$$

............................................

**LABEL** CONTINUE

where values of $K_{b}^{l}$, $K_{e}^{l}$ variables are common for variables $x_{h}^{h}$ and $K_{b}^{r}$, $K_{e}^{r}$ for variables $x_{n}^{n}$.

Consider the case when $\rho_{1}, \ldots, \rho_{n}$ aren't reciprocals. Remember MSCG consists of the several operators.

Let the computations' front of $U$ variable has the form
\[ L_u(k): \delta(p_i + \ldots + p_n i_n) + \Delta_j + r_u = k \]

and for \( V \)
\[ L_v(k): \delta(p_i + \ldots + p_n i_n) + \Delta_j + r_v = k \]

It is evident that computation of \( U \) and \( V \) are carried out for the same \( K \) values in the case when \( r_u = r_v \).

It is not difficult to show that if \( r_v > r_u \) then computation of \( V \) is carried out \( r_v - r_u \) steps later after computation of \( U \) variable (though it can be done in the different points of the domain). While transition to the front of computations with simple parameters the computations of all the variables are carried out for the same \( K \) values. In this case the scheme of the program must be changed.

All the relations are united in the groups.

The relations included in the one group are carried out parallel as it is done before. Though every group begins processing only after the operators of the previous group end it.

Let's formulate the rules of uniting the relations into the one group and the order of the group movement. Uniting of the relations is done in the following way. Let's represent variable \( \Delta_j \) in the form of \( \Delta_j \delta + r_j \). All the relations where \( r_j \) values are equal are united in the one group. Define the order of the groups' movement. Consider the initial front of the wave \( (\delta(p_i + \ldots + p_n i_n) + \Delta = k) \) and determine the minimal value of \( K_{\text{min}} \) for every \( X \) variable. Let's arrange the groups according to the following rule. The first will be the group which includes the relation where \( K_{\text{min}} \) assume the value (on all the values of \( K_{\text{min}}^j \)). Given group unites all the relations where \( r_j = r_0 \). The next group unites the relations where \( r = r_0 + 1 \). If there is no such group then choose the group with \( r = r_0 + 2 \) etc. Note \( r_i < \delta \). If we define the place of the group with \( r = \delta - 1 \) then the next group will be with \( r = 0 \), etc.

**Note.** Let's consider the question if the formulae included into the different groups \( (r_i \neq r_j) \) have the same minimal value \( K_{\text{min}} \), i.e. \( K_{\text{min}}^i = K_{\text{min}}^j \).

Let the wave's fronts for \( X \) and \( X \) have the form:
\[ L_i(k): \delta(p_i + \ldots + p_n i_n + \Delta_j) + r_i = k \]
\[ L_j(k): \delta(p_i + \ldots + p_n i_n + \Delta_j) + r_j = k \]

Assume that \( K_{\text{min}}^i = K_{\text{min}}^j \).

\[ p(i_t' - i_t') + \ldots + p_n(i_t' - i_t') + \delta_j - \delta_j = \frac{r_i - r_j}{\delta} \]

Because \( r_i \neq r_j \) and \( |r_i - r_j| < \delta \) our assumption \( K_{\text{min}} = K_{\text{min}}^i \) isn't right.
The scheme of the program is given below.
\[ K_b = \ldots \]
\[ K_e = \ldots \]

DO 1 K = K_b * K_e

Computations on the relations included into the 1st group
Computations on the relations included into the 2nd group

Computations on the relations included into the \( \delta - 1 \) group

1 CONTINUE

Transition from the computations of the one group to another is sequential. Computations of the relations included into the one group are organized in the way similar to the case of the front with simple parameters i.e. parallel inside each relation and besides each relation separately.

**Examples.**

The examples illustrating given method of loop designing are considered in this part.

Example 1. The domain of changing indexes is a rectangular \( i = 3 \ldots M; \quad j = 3 \ldots N \)

The following relations are set
\[ A_{i,j} = B_{i,j-1} \]
\[ B_{i,j} = A_{i-2,j+2}; B_{i,j-1} \]

The LJP program has the following form:

find \( \rho_1, \rho_2 \) and \( \Delta_B \) where functional \( Z = M * |\rho_1| + N * |\rho_2| \) assumes minimum with the constraints
\[ \rho_2 - \Delta_B > 0 \]
\[ \Delta_B - 2\rho_2 + 2\rho > 0 \]
\[ \rho_2 > 0 \]

The solution is \( \rho_1 = 2, \quad \rho_2 = 1, \quad \Delta_B = 0 \)

Program loop is given below.

\[ K_b = 9 \]
\[ K_e = 2 * M + N \]

DO 1 K = K_b * K_e

\[ DB = \text{MAXX}(3, (K-N)/2) \]
\[ DE = \text{MINN}(M, (K-3)/2) \]

DO 2 D = DB, DE

\[ A(D, K - 2 * D) = B(D, K - 2 * D - 1) \]
2 \quad B(D, K - 2*D) = A(D - 2, K - 2*D + 2); B(D, K - 2*D - 1)

Note. Function \( MAX(X, Y) \) gives us the nearest integer exceeded the greatest of the numbers \( X \) and \( Y \) as a result. Function \( MINN(X, Y) \) gives us the nearest integer which doesn't exceed the least of the numbers \( X \) and \( Y \).

Embedded loop may be carried out for all \( D \) values. This loop corresponds to the computation of the values \( A \) and \( B \) on particular line \( (2*i + j = K) \) with fixed \( K \) value. Transition of the lines (loop on \( K \)) is sequential. Common number of the steps in the loop is \( 2M + N - 8 \).

If we don't solve the LIP task we may first analyse index displacements on each direction. Note that displacements on index \( i \) has one sign. Hence we may not take into consideration link between the variables on this index in those cases when the displacement is non-zero. We can see the result: variable \( B \) doesn't depend on \( A \). We can propose the following scheme of the loop.

\[
\begin{align*}
&\text{DO 1 } i=3,M \\
&\text{DO 2 } j=3,N \\
&A(i,j) = B(i,j-1) \\
&B(i,j) = A(i-2,j+2); B(i,j-1)
\end{align*}
\]

Program loop seems easier in this case. First you mustn't compute the boundaries of changing indexes of the embedded loops (as you must in the case of the first solution). Second the structure of the index expressions is simple enough. Though loop designed in this way is sequential (both on \( i \) and \( j \)). You may do parallel only computations of \( A \) and \( B \) when \( i \) and \( j \) are fixed. Common number of sequential steps is \((M-2) \times (N-2)\) in this case.

Example 2. Let the relations be set

\[
\begin{align*}
X_{i,j} &= f_x(X_{i,j-1}; Y_{i,j}; Z_{i,j}) \\
Y_{i,j} &= f_y(X_{i-1,j}; Y_{i,j}; Z_{i,j-1}) \\
Z_{i,j} &= f_z(X_{i,j-1}; Y_{i-1,j}; Z_{i,j-1})
\end{align*}
\]

the domain of changing indexes

\[
\begin{align*}
i &= 2, \ldots, M \\
j &= 2, \ldots, N
\end{align*}
\]

The fronts of computations for \( X, Y, Z \) are indicated \( L_x(k), L_y(k), L_z(k) \) correspondingly. LIP problem solution has the form:

\[
\begin{align*}
\alpha &= 2, \quad \beta = 6, \quad \Delta_x = 5, \quad \Delta_y = 0, \quad \Delta_z = 3
\end{align*}
\]

The values of \( \alpha \) and \( \beta \) are not reciprocals. The fronts of computations has the form:

\[
L_x(k): \quad -i + 3j + 2 = k \quad r_x = 1
\]
\[ L_j(k): \; -i + 3j = k \; \quad r_j = 0 \]

\[ L_z(k): \; -i + 3j + 1 = k \; \quad r_z = 1 \]

It follows from given above that computations of values \( X \) and \( Z \) are carried out parallel (because \( X \) and \( Z \) are included in the one group and \( Y \) in another one) and sequentially after the computation of \( Y \).

The scheme of the program is given below.

\[
K_b = -M + 6 \\
K_x = -2 + 3N \\
\text{DO \; LABEL \; } K = K_b + K_x \\
DB_x = \text{MAXX}((2 + K) / 3, 2) \\
DE_x = \text{MINN}((M + K) / 3, 2) \\
\text{DO \; LABEL1 \; } DY = DB_x \cdot DE_x \\
\text{LABEL1} \quad Y(3*DY - K, DY) = X(3*DY - K, DY - 1); Y(3*DY - K + 1, DY); \quad Z(3*DY - K - 1, DY - 1) \\
DB_x = \text{MAXX}(K/3, 2) \\
DE_x = \text{MINN}((M + K - 2) / 3, N) \\
\text{DO \; LABEL2 \; } DX = DB_x \cdot DE_x \\
\text{LABEL2} \quad X(3*DX - K + 2, DX) = X(3*DX - K + 3, DX - 1); Y(3*DX - K + 3, DX + 1); \quad Z(3*DX - K + 3, DX) \\
DB_x = \text{MAXX}((K + 1)/3, 2) \\
DE_x = \text{MINN}((M + K - 1)/3, N) \\
\text{DO \; LABEL3 \; } DZ = DB_x \cdot DE_x \\
\text{LABEL3} \quad Z(3*DZ - K - 1, DZ) = X(3*DZ - K, DZ - 1); Y(3*DZ - K, DZ); \quad Z(3*DZ - K + 1, DZ - 1) \\
\text{LABEL \; CONTINUE} \]

Every turn of the loop on \( K \) consists of two sequential steps. At the first step the values of variable \( Y \) in the front's point \(-i + 3j + 1 = k\) are computed parallel, at the second step the values of variable \( X \) in the front's points \(-i + 3j + 2 = k\) and the values of variable \( Z \) in the front's points \(-i + 3j + 1 = k\). The order of computations corresponding components of matrices \( X, Y, Z \) in the case \( M = 7, N = 7 \). E.G. \( X(5, 5) = 27 \) means that the value of variable \( X \) in the point \( (5, 5) \) is computed at 27 step (the first computation is carried out at step 1).

\[
\begin{array}{ccc}
X & Y & Z \\
\end{array}
\]
<table>
<thead>
<tr>
<th></th>
<th>44</th>
<th>43</th>
<th>41</th>
<th>39</th>
<th>37</th>
<th>35</th>
<th>33</th>
<th>31</th>
<th>29</th>
<th>27</th>
<th>25</th>
<th>23</th>
<th>21</th>
<th>19</th>
<th>17</th>
<th>15</th>
<th>13</th>
<th>11</th>
<th>9</th>
<th>7</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>38</td>
<td>36</td>
<td>34</td>
<td>32</td>
<td>30</td>
<td>28</td>
<td>26</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>34</td>
<td>32</td>
<td>30</td>
<td>28</td>
<td>26</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>26</td>
<td>24</td>
<td>22</td>
<td>20</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>20</td>
<td>18</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>19</td>
<td>17</td>
<td>15</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$i \rightarrow j$
Bibliography.

Russian Academy of Science
Keldysh Institute of Applied Mathematics

Andrianov A.N., Andrianova E.A.

Organization of loop process on nonprocedural specification.

Moscow 1996
Introduction.

NORMA is a programming language [1] aimed at automation of mathematical physics problems solutions on parallel computer systems.

The NORMA language allows elimination of a programming phase in transition from formulae specified by a technical expert to a program itself. There is no much difference between formulae and NORMA specifications. In fact these formulae are input data for a translator.

Synthesis of output program is carried out automatically during the translation from NORMA. The order and the way of performing calculations (parallel, vector or sequential) is determined automatically. The order of the language’s sentences is arbitrary (information dependencies are revealed and taken into account during the organization of computing process). There are no such programming terms as memory, loop, control operators in the language. Output program is generated with the architecture of a target computer as a guide.

In fact the program in NORMA is a nonprocedural specification of the problem to be solved. The synthesis of output program raises some mathematical problems but they are solvable in the case of NORMA language.

Some Norma peculiarities makes the process of automatic object program design available for practice realization. They are:

1. Index expressions of calculated variables has the form \( i \pm c \)

where \( i \) - index name, \( c \) - integer constant.

NORMA is a language with single assignment. Any value can be assigned to a variable only once (only once to each point of domain - to the variables defined on domain). The first constraint defines the class of formulae which can be used for the problem’s solution. It isn’t strict in practice as the index expressions of other type are very rare.

Memory allocation and the problems of its economy caused the second constraint. These problems can be solved at the translation stage. The second constraint simplifies the problem of output program synthesis.

The problem of output program is to be solved during the translation. Solving this problem is based on the analysis of the graph of information dependencies. The Most Strongly Connected subGraphs (MSCG) are chosen from the graph. In general case organization of computations for the nodes requires use of special methods.

Assume is a principal operator in NORMA. This operator sets the relations between variables being calculated on a domain. Researching on the subject of computational process organization has been doing for a long time. The notion of local relation on an array was introduced in [5] and called parameter record. Synthesis of loop programs for parameter record in the case of one-dimension array was investigated in [6]. The problems of existence and degree of parallelism of computational process on multidimensional arrays are considered in [7]. Organization of loop computations for relations set on half limited domains is described in [8]. The problems of parallel program synthesis on the base of special formalization of application domain are investigated in [9].
The purpose of this paper is to specify the method of designing loop operators which realizes the relations included into MSCG. Given work is a continuation of [2].

1 FRONT OF COMPUTATIONS.

Executive part in the NORMA language may be represented in the following form:

\[ X^1_i = f_1(X_{i+D_m}, \ldots, X_{i+D_m}) \]

\[ \vdots \]

\[ X^m_i = f_m(X_{i+D_m}, \ldots, X_{i+D_m}) \]

(1)

Here \( I = \{ i_1, \ldots, i_n \} \) denotes the array of indexes corresponded to calculated variables \( X_1, \ldots, X_m \), \( D_{ij} = \{ \delta_{i,j}^1, \ldots, \delta_{i,j}^m \} \) denotes the array of displacements to index variables, \( \delta_{i,j}^k \) - integer constants in NORMA as it is defined in Norma.

Note. The use of one and the same value with different index expressions is possible in the right part of the relations. Besides the independence from some variables is also acceptable. The form of relation's system is used for simplification.

The computations defined by relation are said to be performed in the domain:

\[ M_i^k \leq x_k \leq N_i^k, \quad k = 1, n \]

(2)

The values required for computations and lain outside the domain are considered known. The order of computations isn't set in an explicit way thus information dependencies are to be analysed and then computations are arranged on the result of analysis.

Directed graph is constructed in accordance with the given system of relations. Every relation corresponds to the graph's node. Information dependence of the following type:

\[ X_i^k = f_k(X_{i+D_m}, \ldots) \]

corresponds to the graph's arc from node \( X_i \) to node \( X_k \) with mark \( D_{ki} \).

The most strongly connected subgraphs are searched in the obtained graph and the graph is reduced. A new graph is a non-loop and thus the procedure of arranging nodes according to the relation of "computed before" order is possible. Determination of the loop process for the operators not included into MSCG is easy.

The problem of loop process determination for the operators included into MSCG will be considered below.

At first we consider the methods of computations sequence informally.

Definition 1. Set of the points satisfied equation

\[ \Lambda(X, k) = \{ i_1, \ldots, i_n \}: \quad p_1 i_1 + \ldots + p_n i_n + \Delta^i = k, \quad (3) \]

are called basically front and equation (3) - basical equation of \( X \) variable at \( k \) step computations' front.
Note. Difference \( \Delta_{i} - \Delta_{i} \), \( t, l = \overline{t}, \overline{m} \) defines several steps of computations of variable \( \dot{X} \) "delay" (outstrip") from the computations of variable \( \dot{X} \) in the same point.

Note that the range of changing step \( k \) value is defined by all means for the initial domain (2). Let the range of step \( k \) changing has the form: \([k_{\text{min}}, k_{\text{max}}]\). Computation of the variable’s values consists of the sequence of steps in the initial domain. At first value \( k_{\text{min}} \) is assigned to variable \( k \): \( k = k_{\text{min}} \) and the values of variable \( \dot{X} \) are computed in all the points of basal front (3). Then start with the next step. 1 is added to \( k \) value \((k = k + 1)\) and the value of \( \dot{X} \) is calculated on a new front. Given process is continued until \( k \) value assumes its maximum \( k_{\text{max}} \).

Note. Every variable \( \dot{X} \) (when value of \( k \) step is fixed) is computed on the plane generally parallel to the plane where the value of \( \dot{X} \) is computed.

Definition 2. Basical equation where the Greatest Common Devisor (GCD) \((p_{1}, ..., p_{n}) = 1\) is called reduced basical equation.

Let’s consider the following system of equations:

\[
\begin{align*}
\Lambda_{1}(\dot{X}, k_{1}): \rho_{1_{i}i_{1}} + ... + \rho_{n_{i}n_{1}} + \Delta'_{1} &= k_{1} \\
\Lambda_{2}(\dot{X}, k_{2}): \rho_{2_{i}i_{1}} + ... + \rho_{n_{i}n_{2}} + \Delta'_{2} &= k_{2} \\
&\cdots \\
\Lambda_{r}(\dot{X}, k_{r}): \rho_{s_{i}i_{s}} + ... + \rho_{n_{i}n_{r}} + \Delta'_{r} &= k_{r}
\end{align*}
\]

(4)

Here \( \rho_{s_{i}}, \Delta'_{s}, j = \overline{1}, n \), \( s = \overline{1}, r \), \( t = \overline{1}, \overline{m} \) are integer numbers.

The first equation from system (4) is basical.

Definition 3. Front of computations of \( \dot{X} \) value at \( k_{1} \) step is a set of the points satisfied basical equation, the order of the points’ traversal is specified by the equations (4).

The order of variables’ computations is considered below. The values of variable \( \dot{X} \) in the opines with integer coordinates belonged to the plane set by the basical equation of the front are to be calculated when \( k_{1} \) (from range \([k_{\text{min}}, k_{\text{max}}]\)) is fixed. The order of other points’ traversal is specified by other equations. The second equation defines the family of \( n \)-dimensional hyperplanes (according to \( k_{2} \) values changing from \( k_{2_{\text{min}}} \) to \( k_{2_{\text{max}}} \)). By adding the first equation we obtain the family of \( n-1 \)-dimensional hyperplanes which cover \( n \)-dimensional hyperplane set by the basical equation. Thus the points on the \( n-1 \)-dimensional hyperplane are calculated first (when \( k_{2} = k_{2_{\text{min}}} \)). Then 1 is added to \( k_{2} \) value and the values in the points of the next \( n-1 \)-dimensional plane are calculated etc. Let’s consider
fixed \(n-1\)-dimensional hyperplane (defined by the first two equations when \(k_1 = \overline{k}_1 \) and \(k_2 = \overline{k}_2\)). The order of the traversal of the points belonged to this hyperplane is determined in the same way: the third equation is considered and varying \(k_3\) values from \(k_3^{\text{min}}\) to \(k_3^{\text{max}}\) we obtain the family of hyperplanes covering all the points of \(n-1\)-dimensional hyperplane. The third equation with the equations defined \(n-1\)-dimensional hyperplane sets \(n-2\)-dimensional hyperplane (when \(k_3 = \overline{k}_3\) is fixed). If there are no other equations then the order of the traversal of the points on this hyperplane is arbitrary (i.e. parallel).

In the other case the next equation is considered etc. This procedure will be continue until the last equation is considered. It is evident that if \(r=n\) then the traversal of the points of the initial domain will be sequential i.e. point by point. If \(r<n\) the computation may be carried out parallel in the points \(n-r+1\)-dimensional hyperplane. In particular if the front of computation consists of the one (basical) equation then the computations of the given variable may be carried out independently in all the points of the plane.

Note. The questions connected with the determination of the front of computations are described in [3-4].

2 Computations on the basical front

Let's consider the problem of the order's determination of the initial domain's points' traversal when the front of computation is set by the basical equation:

\[
\Lambda (X, k) = \{ i_1, ..., i_n \} : p_i^1 + ... + p_n^i + \Delta'_i = k \quad (5)
\]

Denote \(\sigma_i = GCD (p_i, ..., p_n)\). Thus every coefficient of the equation \(p_i, \quad i = 1, n\) is expressed in the following way: \(p_i = \sigma_i \cdot \overline{p}_i\). Represent equation (5) in the form:

\[
\sigma_i \cdot (\overline{p}_i i_1 + ... + \overline{p}_n i_n) + \Delta'_i = k \quad (6)
\]

Denote \(\overline{p}_i i_1 + ... + \overline{p}_n i_n = t_i\). Then equation (6) assumes the form:

\[
\sigma_i t_i - k = -\Delta'_i
\]

Write the solution of this equation:

\[
\begin{align*}
t_i &= d_i^l \\
k_i &= \Delta'_i + \sigma_i d_i^l
\end{align*}
\]

Parameter \(d_i^l\) is used for a sequential examination of all the planes covering the initial domain. As the parameter has been obtained for every variable then let's use the one which is common for all the variables. From \(k' = k\), \((l, t = 1, m)\) follows:

\[
d_i^l = d_i^l + \frac{\Delta'_i - \Delta_i}{\sigma_i}
\]

Let \(d_i^{\bar{l}} = \max_{l, m} \{d_i^l\}\)
Denote \( R_{kh}^l = \frac{\Delta^l - \Delta^i}{\sigma_1} \) and represent \( R_{kh}^l \) in the form

\[
R_{kh}^l = [R_{kh}^l] + \{R_{kh}^l\}
\]

where \([R_{kh}^l]\) - integer part of a number and \(\{R_{kh}^l\}\) - its fractional part.

Express all \(d_i^l\) (\(l = 1, m\)) through \(\tilde{d}_i^{k_1}\):

\[
d_i^l = \tilde{d}_i^{k_1} - [R_{kh}^l]
\]

Parameter \(d_i^l\) can assume only integer values. Sequence of \(k\) step’s initial value must be kept in the transition to the common parameter. Thus variable \(r_i^l\) is to be linked with every \(X^l\):

\[
r_i^l = \{R_{kh}^l\}
\]

Now one and the same parameter \(\tilde{d}_i^{k_1}\) is used for the sequential examination of all the planes. When \(\tilde{d}_i^{k_1}\) is fixed the computation of variables are arranged on increasing of \(r_i^l\) values.

Define the range of \(\tilde{d}_i^{k_1}\) values changing for every \(X^l\):

\[
\begin{align*}
d_{i,1}^{\text{min}} &= \sum_{p_i > 0} p_i M_i^l + \sum_{p_i < 0} p_i N_i^l + [R_{kh}^l] \\
d_{i,1}^{\text{max}} &= \sum_{p_i > 0} p_i N_i^l + \sum_{p_i < 0} p_i M_i^l + [R_{kh}^l]
\end{align*}
\]

Common range of \(\tilde{d}_i^{k_1}\) parameter’s changing is a segment \([d_{i,1}^{\text{min}}, d_{i,1}^{\text{max}}]\) where

\[
d_{i,1}^{\text{min}} = \min_{l=1, m} \{d_{i,1}^{\text{min}}\}, \quad d_{i,1}^{\text{max}} = \max_{l=1, m} \{d_{i,1}^{\text{max}}\}
\]

The traversal of the initial domain’s points is carried out by the sequential examination of the planes. Transition from the examination of the one plane to another is performed in the loop changing values of loop parameter \(\tilde{d}_i^{k_1}\) from \(d_{i,1}^{\text{min}}\) to \(d_{i,1}^{\text{max}}\). Range \((d_{i,1}^{\text{min}}, d_{i,1}^{\text{max}})\) is common for all the variables. As every variable \(X^l\) has its own range of computation steps \((d_{i,1}^{\text{min}}, d_{i,1}^{\text{max}})\) the test of the type given below is placed before the computation of the variable at \(\tilde{d}_i^{k_1}\) step:

IF \(\tilde{d}_i^{k_1} \in (d_{i,1}^{\text{min}}, d_{i,1}^{\text{max}})\) THEN ... 

Let’s consider the question connected with the determination of the points with integer coordinates on the plane set by the following equation:

\[
\tilde{p}_1 l_1 + \ldots + \tilde{p}_n l_n + [R_{kh}^l] = \tilde{d}_i^{k_1}
\]

Definition 4. Further the transition from the basical equation (5) to the reduced basical equation (10) will be called algorithm of reduction.

Represent equation (10) in the following form:
\[\sigma_2(p_1^{l_1} + \ldots + p_{i_n-1}^{l_{i_n-1}}) + p_{i_n} = \bar{d}_1^{k_1} - [R_1^{l_1}] \]  

(11)

where \( \sigma_2 = GCD(\bar{p}_1, \ldots, \bar{p}_{i_n-1}) \). Denote \( t_2 = p_1^{l_1} + \ldots + p_{i_n-1}^{l_{i_n-1}} \). In this case the equation has the following solution:

\[
t_2 = (\bar{d}_1^{k_1} - [R_1^{l_1}])t_2^0 \pm p_{i_n}d_2
\]
\[
i_n = (\bar{d}_1^{k_1} - [R_1^{l_1}])i_n^0 \pm \sigma_2 d_2,
\]

where \( t_2^0 \) and \( i_n^0 \) are the solution of \( \sigma_2 t_2 + p_{i_n} i_n = 1 \) equation.

Note. Choice of the sign is arbitrary in this case. The domain of changing of the values of \( t_2 \) variable is determined by the following range: \( t_2 \in (t_2^{\min}, t_2^{\max}) \) where

\[
t_2^{\min} = \sum_{p_1^{l_1} > 0} p_1^{l_1} M_i^{l_1} + \sum_{p_1^{l_1} < 0} p_1^{l_1} N_i^{l_1},
\]
\[
t_2^{\max} = \sum_{p_1^{l_1} > 0} p_1^{l_1} N_i^{l_1} + \sum_{p_1^{l_1} < 0} p_1^{l_1} M_i^{l_1}
\]

Taking into consideration the domain of \( t_2 \) variable changing and the boundaries of index variable \( i_n \) changing the range of parameter \( d_2 \) changing can be defined.

Then we consider the following equation:

\[p_1^{l_1}i_1 + \ldots + p_{i_n-1}^{l_{i_n-1}} = (\bar{d}_1^{k_1} - [R_1^{l_1}])t_2^0 \pm p_{i_n}d_2\]  

(12)

Denote the right part by \( T_1 \). Let \( \sigma_3 = GCD(p_1^{l_1}, \ldots, p_{i_n-2}^{l_{i_n-2}}) \). Represent equation (12) in the form:

\[\sigma_3(\bar{p}_1^{l_1}i_1 + \ldots + \bar{p}_{i_n-2}^{l_{i_n-2}}i_{n-2}) + p_{i_n-1}^{l_{i_n-1}} = T_1 \]

and denoting \( T_3 = \bar{p}_1^{l_1}i_1 + \ldots + \bar{p}_{i_n-2}^{l_{i_n-2}}i_{n-2} \) write the solution of the last equation:

\[
t_3 = T_3^{\min} \pm p_{i_n-1} d_3
\]
\[
i_{n-1} = T_3^{\max} \pm \sigma_3 d_3
\]

Variables \( t_3^0 \) and \( i_{n-1}^0 \) are the solution of the equation \( \sigma_3 t_3 + p_{i_n-1}^{l_{i_n-1}} i_{n-1} = 1 \). The range of \( d_3 \) parameter changing is determined in the same way as for parameter \( d_2 \).

Then the process of solving will continue till the following equation is obtained:

\[p_1^{l_1}i_1 + p_2^{l_2}i_2 = T_{i_n-2}\]

Its solution has the form:

\[
i_1 = T_{i_n-2}^{\min} \pm p_2^{l_2}d_n
\]
\[
i_2 = T_{i_n-2}^{\max} \pm p_1^{l_1}d_n
\]

Thus the traversal's scheme of the initial domain's points (2) when the from of computation (5) is set has the following form:
\[ d_{1}^{\text{min}} = \ldots \]
\[ d_{1}^{\text{max}} = \ldots \]
\[ d_{2}^{\text{min}} = \ldots \]
\[ d_{2}^{\text{max}} = \ldots \]
\[ \text{DO 1 } \quad d_i = d_{i}^{\text{min}}, d_{i}^{\text{max}} \]
\[ d_{n}^{\text{min}} = \ldots \]
\[ d_{n}^{\text{max}} = \ldots \]
\[ \text{DO 1 } \quad d_i = d_{i}^{\text{min}}, d_{i}^{\text{max}} \]
\[ \text{IF } \quad d_i \in (d_{i}^{\text{min}}, d_{i}^{\text{max}}) \& \ldots \& d_i \in (d_{i}^{\text{min}}, d_{i}^{\text{max}}) \quad \text{THEN} \]
\[ C \quad \text{The expressions of the index variables are omitted.} \]
\[ X(...)=\ldots \]
\[ \text{IF } \quad d_i \in (d_{i}^{\text{min}}, d_{i}^{\text{max}}) \& \ldots \& d_i \in (d_{i}^{\text{min}}, d_{i}^{\text{max}}) \quad \text{THEN} \]
\[ X(...)=\ldots \]
\[ \text{CONTINUE} \]

Note 1. Sequence of the formulae is determined based on the parameters \( \xi' \) values.

Note 2. The case when the values of some coefficient \( \rho_k = 0 \) corresponds to a loop along the direction \( i_k \) with initial value \( M_k \) and the final value \( N_k \).

3 Computation on the front of common type

An algorithm of loop parameters' determination for the fronts of computations set by the system of equations (4) is considered. At first we prove a theorem. We'll need its result in the further work.

**Theorem 1.** There exists matrix \( A_n(\rho_1, \ldots, \rho_n) \) with integer elements (dimension \( n \times n \)). The first row of the matrix and its \( \det A_n(\rho_1, \ldots, \rho_n) = 1 \) consist of the coefficients of the basical equation.

**Proof.** The proof will be performed by induction. For \( n = 2 \) matrix has the form:
\[ A_2(p_1, p_2) = \begin{pmatrix} p_1 & p_2 \\ \beta & \alpha \end{pmatrix} \]

where \( \alpha \) and \( \beta \) are the solution of \( p\alpha - p\beta = 1 \) equation.

Let's assume that everything is right for \( n < k \). Prove it for \( n = k \). Represent the resulting matrix in the form of two matrixes' multiplication: \( A_n = B_n \cdot C_n \) where matrix \( B_n \) has the form:

\[
B_n = \begin{pmatrix} a & 0 & \ldots & 0 & p_n \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ \alpha & 0 & \ldots & 0 & \beta \end{pmatrix}
\]

where \( a = GCD(p_1, \ldots, p_{n-1}) \) and \( \alpha, \beta \) are the solution of equation:

\[ a\beta - p_\alpha = 1 \]

Note that \( \det B_n = a\beta - p_\alpha = 1 \).

Matrix \( C_n \) has the following form:

\[
C_n = \begin{pmatrix} A_{n-1}(\varphi_1, \ldots, \varphi_{n-1}) & 0 \\ \vdots & \vdots \\ 0 & \ldots & 0 & 1 \end{pmatrix}
\]

where \( \varphi_i = \frac{p_i}{a}, \quad i = 1, n-1 \).

Matrix \( A_{n-1}(\varphi_1, \ldots, \varphi_{n-1}) \) exists by the induction assumption.

\[ \det C_n = \det A_{n-1}(\varphi_1, \ldots, \varphi_{n-1}) = 1 \]

Hence

\[ \det A_n = \det B_n = \det C_n = 1 \]

Matrix \( A_n \) is integer as matrixes \( B_n \) and \( C_n \) are integer. Let's consider the first line of the matrix \( A_n(p_1, \ldots, p_n) \). The elements of the first line are calculated on the formula:

\[
a_{i,1} = \sum_k b_{i,k} c_{i,1} = a\varphi_i = p_i \\
a_{n,1} = p_n.
\]

Now let's start considering the method of loop parameters' determination.

Front of computations for \( \mathcal{X} \quad l = 1, m \) variables is considered.
\[ p_{i_1,t_1} + \ldots + p_{i_n,t_n} + \Delta' = K_i \]

\[ p_{i_1,t_1} + \ldots + p_{i_n,t_n} + \Delta' = K_r \]

Denote \( \sigma_1 = NOD(p_{i_1}, \ldots, p_{i_n}) \). Using the algorithm of reduction in 2 we transit to the following equations:

\[ p_{i_1,t_1} + \ldots + p_{i_n,t_n} + [R_{k_1}^i] = \tilde{d}_i^{k_1} \]

\[ p_{i_1,t_1} + \ldots + p_{i_n,t_n} + \Delta' = K_r \]

where \( [R_{k_1}^i] = \frac{\Delta_i - \Delta'_i}{\sigma_1} \) and \( p_{i_j} = \frac{p_{i_j}}{\sigma_1} \). Perform the following coordinate's transformation:

\[ A_n(p_{i_1}, \ldots, p_{i_n}) \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} = \begin{pmatrix} i_n^0 \\ \vdots \\ i_n^0 \end{pmatrix} \]

Matrix \( A_n \) is chosen according to theorem 1, where the first equation from (14) is chosen according to theorem 1 where the first equation from (14) is a reduced basic equation and the first row in matrix \( A_n \). Denote:

\[ (p_{i_1}, \ldots, p_{i_n}) = (p_{i_1}, \ldots, p_{i_n})A_n^{-1} \quad k = \overline{2}, r \]

We'll consider the following system of equations instead of system of equations (13):

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + p_{i_1}^{\prime} (\tilde{d}_i^{k_1} - [R_{k_1}^i]) + \Delta'_2 = K_2 \]

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + \Delta'_2 = K_r \]

where we use the fact that \( i_1^0 = p_{i_1,t_1} + \ldots + p_{i_n,t_n} = \tilde{d}_i^{k_1} - [R_{k_1}^i] \). Using algorithm of reduction we transit from the system of equations (15) to the system of equations:

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + [R_{k_1}^i] = \tilde{d}_i^{k_2} \]

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + [R_{k_1}^i] = K_2 \]

\[ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \]

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + \Delta'_3 = K_3 \]

\[ p_{i_2,t_2} + \ldots + p_{i_n,t_n} + \Delta'_3 = K_r \]

where the expression for \( R_{k_2}^i \) has the form:

\[ R_{k_2}^i = \frac{\Delta_i^2 - \Delta'_2 + p_{i_1}^{\prime} ([R_{k_1}^i] - [R_{k_2}^i])}{\sigma_2} \]

Thus we have the system of equations similar to (14) differed only in smaller number of equations and index variables. Further we'll continue this process till we get the equation of the following type:
\[
\begin{align*}
\mathcal{P}^{n-r+2}_r i_r^{n-r+2} + \ldots + \mathcal{P}^{n-r+2}_n i_n^{n-r+2} + [R_{i_{k_i}}^n] &= q^k_i \\ 
\text{sign} \left( \mathcal{P}^{n-r+2}_r i_r^n + [R_{i_{k_i}}^n] \right) &= d^n_i 
\end{align*}
\]

$r < n$

$r = n$

The methods of derivation of index expressions for such an equation is given in 2. We obtain the values for indexes: \( i^{n-r+2}_r, \ldots, i^{n-r+2}_n \) (for the case \( r < n \)) as a result of the last equation's solution. Now we carry out the following sequence of back transformations for the initial indexes' derivation:

\[
\begin{pmatrix}
i_r^{n-r+3} \\
\vdots \\
i_n^{n-r+3}
\end{pmatrix}
= A^{-1}_{n-r+2}
\begin{pmatrix}
i_r^{n-r+2} \\
\vdots \\
i_n^{n-r+2}
\end{pmatrix}
\]

Further we carry out the following transformation using the fact that \( i^{n-r+3}_r = d^{k-1}_{r-1} - [R_{i_{k_i}}^{r-1}] \):

\[
\begin{pmatrix}
i_r^{n-r+4} \\
\vdots \\
i_n^{n-r+4}
\end{pmatrix}
= A^{-1}_{n-r+3}
\begin{pmatrix}
d^{k-1}_{r-1} - [R_{i_{k_i}}^{r-1}] \\
\vdots \\
i_n^{n-r+3}
\end{pmatrix}
\]

Thus process will be continued till the values for the initial indexes of the task are obtained:

\[
\begin{pmatrix}
i_1 \\
\vdots \\
i_n
\end{pmatrix}
= A^{-1}_n
\begin{pmatrix}
d^{k_1}_{i_{k_i}} - [R_{i_{k_i}}^n] \\
\vdots \\
i_n^n
\end{pmatrix}
\]

4

The fronts of computations in 3-dimensional domain

To illustrate method of loop parameters determination given above the fronts for the variables defined on 3-dimensional domain are considered. The front of computations set by two equations are considered in the first part. The front of computations is set by three equations in the second part.

4.1 Sequential-parallel computations

Let's assume that the front for some variable is set by two following equations:

\[
\begin{align*}
\mathcal{P}_1 i_1 + \mathcal{P}_2 i_2 + \mathcal{P}_3 i_3 + \Delta_i &= K_1 \\
\mathcal{P}_2 i_1 + \mathcal{P}_2 i_2 + \mathcal{P}_3 i_3 + \Delta_2 &= K_2
\end{align*}
\]

Basical equation defines the points of the plane where requested value of the variable is to be calculated. The second equation specifies the order of the points' traversal on the plane set by the first equation.

Informally it means the following things. The traversal of the points with integer coordinates belonged to the plane set by the first equation is to be organized. The order of the traversal is following. As the values of the points' coordinate must satisfy both equations (17) (that is the setting of the line in the space if \( K_2 \) and \( K_1 \) values are fixed) then really we need to organise the traversal of the points with integer coordinates along the set line. The computations of variables' values on such a
line may be carried out parallel. Our purpose is to find such expressions for indexes’ values \((i_1, i_2, i_3)\) (which are the expressions of some parameters \(a_1, a_2, a_3\)) that the value of \(K_2\) variable increases when the values of parameters \(a_1, a_2, a_3\) increase. Then increasing the value of \(K_2\) variable we start the traversal of the points along the next line etc.

Denote \(\sigma_1 = GCD(p_{1,1}, p_{1,2}, p_{1,3})\). Using algorithm of reduction transit from the system (17) to the following system of equations:

\[
\begin{align*}
\bar{p}_{1,1} i_1 + \bar{p}_{1,2} i_2 + \bar{p}_{1,3} i_3 + \lfloor \frac{R_{1,k}}{\alpha} \rfloor &= K_1 \\
\bar{p}_{2,1} i_1 + \bar{p}_{2,2} i_2 + \bar{p}_{2,3} i_3 + \Delta_2' &= K_2
\end{align*}
\] (18)

Denote: \(a = GCD(\bar{p}_{1,1}, \bar{p}_{1,2})\). Let’s assume that \(\bar{p}_{1,1} = a\phi_1\) and \(\bar{p}_{1,2} = a\phi_2\). Note that \(GCD(\phi_1, \phi_2) = 1\) and \(GCD(a, \bar{p}_{1,3}) = 1\). Denote by \(\sigma\) and \(\gamma\) the solutions of the equation: \(\phi_1\sigma - \phi_2\gamma = 1\) and by \(\beta\) and \(\alpha\) solutions of equation: \(a\beta - \bar{p}_{1,3}\alpha = 1\).

Consider the following matrix:

\[
A_3 = \begin{pmatrix}
 a\phi_1 & a\phi_2 & \bar{p}_{1,3} \\
 \gamma & \sigma & 0 \\
 \alpha\phi_1 & \alpha\phi_2 & \beta
\end{pmatrix}
\]

Transform the initial coordinates:

\[
A_3(a\phi_1, a\phi_2, \bar{p}_{1,3}) \begin{pmatrix}
 i_1 \\
 i_2 \\
 i_3
\end{pmatrix} = \begin{pmatrix}
 \bar{p}_1' \\
 \bar{p}_2' \\
 \bar{p}_3'
\end{pmatrix}
\] (19)

and denote \((\bar{p}_1'^3, \bar{p}_2'^3, \bar{p}_3'^3) = (p_{2,1}, p_{2,2}, p_{2,3}) \cdot A_3^{-1}\). The second equation assumes the form:

\[
\bar{p}_{2,1} i_1^3 + \bar{p}_{2,2} i_2^3 + \bar{p}_{2,3} i_3^3 + \Delta_2' = K_2
\]

As \(i_1^3 = \bar{d}_1^{k}\) then the second equation from (18) assumes the form:

\[
\sigma_2(\bar{p}_{2,1}' i_1^3 + \bar{p}_{2,2}' i_2^3) - K_2 = -\bar{p}_{2,1}(\bar{d}_1^k - \lfloor \frac{R_{1,k}}{\alpha} \rfloor) - \Delta_2
\] (20)

where \(\sigma_2 = GCD(p_{2,1}^3, p_{2,2}^3)\). Rewrite equation (20) in the form:

\[
\sigma_2 t_i - K_2 = -\bar{p}_{2,1}(\bar{d}_1^k - \lfloor \frac{R_{1,k}}{\alpha} \rfloor) - \Delta_2
\] (21)

where \(t_i = p_{2,2}^3 i_2^3 + p_{2,3}^3 i_3^3\). The solution of this equation has the form:

\[
t_i = \bar{d}_2^k \\
K_2' = \bar{p}_{2,1}(\bar{d}_1^k - \lfloor \frac{R_{1,k}}{\alpha} \rfloor) + \Delta_2 + \sigma_2\bar{d}_2^k
\]
Then all the parameters $d^i_k$ will be reduced to the common parameter $\tilde{d}^k_2$ and we start solving the problem:

$$p_{2,2}^2 + p_{2,3}^2 + [R^2_{i_k}] = \tilde{d}^k_2$$

Write the solution of this equation:

$$\tilde{p}_2^2 = (\tilde{d}^k_2 - [R^2_{i_k}]) \frac{p^2_2 + p^2_3}{p^2_2 + p^2_3} d^i_2 \quad (22)$$

Note. The choice of the sign before parameter $d^i_k$ is arbitrary. It corresponds to the fact that the traversal of the points is done by the sequential examination of the lines but the traversal along every line is done parallel on every point.

The explanations of some notations' usage is given below. In this case $\tilde{p}_2^2$ and $\tilde{p}_3^2$ are the solutions of Euclid's equation: $p_{2,2}^2 + p_{2,3}^2 = 1$. The variable $R^2_{i_k}$ has the form:

$$R^2_{i_k} = \frac{p_{2,1}^2([R^1_{i_k}]) + \Delta^k_2 - \Delta_2}{\sigma_2}$$

The final solution is evident:

$$\begin{pmatrix}
  \tilde{d}_1 \\
  \tilde{d}_2 \\
  \tilde{d}_3
\end{pmatrix} = A_5^{-1}
\begin{pmatrix}
  \tilde{d}^k_2 - [R^2_{i_k}] \\
  (\tilde{d}^k_2 - [R^2_{i_k}]) \frac{p^2_2 + p^2_3}{p^2_2 + p^2_3} d^i_2 \\
  (\tilde{d}^k_2 - [R^2_{i_k}]) \frac{p^2_2 + p^2_3}{p^2_2 + p^2_3} d^i_3
\end{pmatrix}$$

### 3.2 Sequential computations

Let the front of some variables be set by three following equations:

$$\begin{align*}
p_{1,1} + p_{2,2} + p_{3,3} + \Delta_i &= K_1 \\
p_{2,1} + p_{2,2} + p_{3,3} + \Delta_2 &= K_2 \\
p_{3,1} + p_{2,2} + p_{3,3} + \Delta_3 &= K_3
\end{align*} \quad (23)$$

The front of computations set in this form determines strict sequential order of computation of the variables on the plane set by the basical equation. The first two equations set the family of the lines (when $K_2$ varies from $K_2^{min}$ to $K_2^{max}$) as it was in the previous case. At first the values of the variables in the points of the first line (when $K_2 = K_2^{min}$) then increasing the value of $K_2$ by 1 we start the traversal of the points on the next line etc. The traversal of the points along every line was arbitrary in the previous case. When the front of computations is set by three equations the last one determines the direction of the traversal of the points along the lines.

As it was in the previous case instead of system of equations (23) we start considering the following ones:
\[
\begin{align*}
\bar{p}_{1,1i} + \bar{p}_{1,2i} + \bar{p}_{1,3i} + [R_{ik}] &= \bar{d}_{1i}^k \\
p_{2,1i} + p_{2,2i} + p_{2,3i} + \Delta_2^i &= K_2 \\
p_{3,1i} + p_{3,2i} + p_{3,3i} + \Delta_3^i &= K_3
\end{align*}
\] (24)

Using the transformation of the coordinates (19) we obtain instead of the second and the third equations the following ones:
\[
\begin{align*}
p_{2,2i}^3 + p_{2,3i}^3 + p_{2,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + \Delta_2^i &= K_2 \\
p_{3,2i}^3 + p_{3,3i}^3 + p_{3,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + \Delta_3^i &= K_3
\end{align*}
\] (25)

Use the algorithm of reduction and obtain the following system of equations:
\[
\begin{align*}
\bar{p}_{2,2i}^3 + \bar{p}_{2,3i}^3 + [R_{ik}^2] &= \bar{d}_{2i}^k \\
p_{3,2i}^3 + p_{3,3i}^3 + p_{3,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + \Delta_3^i &= K_3
\end{align*}
\] (26)

Then using the transformation of the coordinates:
\[
A_2 \begin{pmatrix} i_2^2 \\ i_3^2 \end{pmatrix} = \begin{pmatrix} \bar{p}_{2,2}^3 & \bar{p}_{2,3}^3 \\ \beta & \alpha \end{pmatrix} \begin{pmatrix} i_2^3 \\ i_3^3 \end{pmatrix} = \begin{pmatrix} i_2^2 \\ i_3^2 \end{pmatrix},
\]
where \(\beta\) and \(\alpha\) are the solutions of equation \(\bar{p}_{2,2i}^3 \alpha - \bar{p}_{2,3i}^3 \beta = 1\) and denoting
\[
(p_{2,2i}^3, p_{2,3i}^3) = (p_{2,2i}^3, p_{2,3i}^3) \cdot A_2^{-1}
\]
we obtain the second equation from (26) in the following form:
\[
p_{3,2i}^3 + p_{3,3i}^3 + p_{3,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + \Delta_3^i = K_3
\]

As \(i_2^3 = \bar{d}_{2i}^k - [R_{ik}^2]\) we may rewrite the last equation in the following form:
\[
p_{3,2i}^3 + K_3 = -p_{3,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) - p_{3,2i}^3 (\bar{d}_{1i}^k - [R_{ik}]) = \Delta_3^i
\]
The solution of this equation has the form:
\[
\begin{align*}
i_2^3 &= \text{sign}(p_{2,3i}^3) d_{3i}^2 \\
K_3' &= p_{3,1i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + p_{3,2i}^3 (\bar{d}_{1i}^k - [R_{ik}]) + \Delta_3^i + |p_{3,3i}^3| d_{3i}^2
\end{align*}
\]
Reduced to common parameter \(d_{3i}^k\) we obtain the solution for \(i_2^3\) in the form:
\[
i_2^3 = \text{sign}(p_{2,3i}^3) (\bar{d}_{3i}^k - [R_{ik}])
\]

Now we start the procedure of the initial indexes' derivation. At first we transform:
\[
\begin{pmatrix} i_2^3 \\ i_3^3 \end{pmatrix} = A_2^{-1} \begin{pmatrix} \bar{d}_{2i}^k - [R_{ik}^2] \\ \text{sign}(p_{2,3i}^3) (\bar{d}_{3i}^k - [R_{ik}]) \end{pmatrix}
\]
and then:
\[
\begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} = A_3^{-1} \begin{pmatrix} d_{1i}^k - [R_{ik}] \\ i_2^2 \\ i_3^2 \end{pmatrix}
\]
THE EXAMPLE OF LOOP OPERATORS’ CONSTRUCTION

We consider the example of loop operator’s construction for the following system of relations:

\[
\begin{align*}
X_{i,j,k} &= f_x(X_{i,j-1,k}, Y_{i,j,k}) \\
Y_{i,j,k} &= f_y(X_{i-1,j,k}, Y_{i,j+1,k}, U_{i-1,j,k}, Z_{i,j,k}) \\
Z_{i,j,k} &= f_z(Y_{i,j,k+1}, Z_{i,j-1,k}) \\
U_{i,j,k} &= f_u(Y_{i,j,k+1}, U_{i,j,k+1}, V_{i-1,j,k}) \\
V_{i,j,k} &= f_v(U_{i,j+1,k}, V_{i,j,k-1})
\end{align*}
\]

Set values are to be calculated in the domain set by the system of inequalities:

\[i \in (1,10); \quad j \in (1,10); \quad k \in (1,10).\]

For \(X, Y, Z, U\) and \(V\) variables the following equations of the computations’ fronts are obtained:

\[
\begin{align*}
X: \quad 2i + 1 &= K_1, \quad j = K_2 \\
Y: \quad 2i &= K_1, \quad -2k + 1 = K_2, \quad -j = K_3 \\
Z: \quad 2i &= K_1, \quad -2k = K_2, \quad j = K_3 \\
U: \quad 2i + 1 &= K_1, \quad -k = K_2 \\
V: \quad 2i + 2 &= K_1, \quad k = K_2
\end{align*}
\]

Using the algorithm of reduction we obtain the following systems of computations’ fronts.

\[
\begin{align*}
X: \quad i &= K_1, \quad j = K_2 \\
Y: \quad i &= K_1, \quad -2k + 1 = K_2, \quad -j = K_3 \\
Z: \quad i &= K_1, \quad -2k = K_2, \quad j = K_3 \\
U: \quad i &= K_1, \quad -k = K_2 \\
V: \quad i + 1 &= K_1, \quad k = K_2
\end{align*}
\]

Here the values of parameters \(r^l_1, l = X, Y, Z, U, V\) has the following form:

\[
r^x_1 = 1, \quad r^y_1 = 0, \quad r^z_1 = 0, \quad r^u_1 = 1, \quad r^v_1 = 0.
\]

It is easy to prove that the range of parameter \(d_i\) changing is \((1,11)\). Besides the range for every value (except \(V\) value) is \((1,10)\) and for \(V\) - \((2,11)\).
DO 1  d_i = 1,11
        IF  d_i \in (1,10)    IF  d_i \in (2,11)
        \[
        \begin{array}{c|c|c}
        Y & Z & V \\
        \end{array}
        \]
        C
          ----------------------------------------
        C  X  U
        C  Y, Z, V,
        C  r_1^X = r_1^U = 1
        IF  d_i \in (1,10)    IF  d_i \in (1,10)
        \[
        \begin{array}{c|c|c}
        X & U \\
        \end{array}
        \]
        1  CONTINUE

Determination of loop operator’s parameters for $X, U$ and $V$ values isn’t difficult. The fronts of computations of this variables are different thus they are considered separately. The procedure of reduction to the common parameter isn’t necessary. Thus we’ll consider one of these variables.

The front of computations for variable $X$ has the following form:

\[
\begin{align*}
    i &= d_1 \\
    j &= K_2
\end{align*}
\]

Matrix of transformation in this case is a unit one. Change the system of notations:

\[
A_3
\]

\[
A_3 \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} i' \\ j' \\ k' \end{pmatrix}
\]

Corresponds to (20) the second equation assumes the form:

\[
\tilde{j} = d_2.
\]

The form of the index expressions follows from (22):

\[
\begin{align*}
    \tilde{j} &= d_2 \\
    \tilde{k} &= \pm d_3
\end{align*}
\]

Transiting to the initial notations:

\[
\begin{pmatrix} i \\ j \\ k \end{pmatrix} = A_3^{-1} \cdot \begin{pmatrix} d_1 \\ d_2 \\ \mp d_3 \end{pmatrix}
\]

We obtain the solution:

\[
\tilde{j} = d_2.
\]
\[ \bar{j} = d_2 \]
\[ \bar{k} = \pm d_3 \]

Thus the scheme of \( X \) variable computation has the following form:

\[
\begin{align*}
\text{DO 2 } d_2 & = 1,10 \\
\text{DO 2 } d_3 & = 1,10 \\
X(d_1,d_2,d_3) & = ...
\end{align*}
\]

\[ 2 \text{ CONTINUE} \]

Nested loop (on \( d_3 \)) may be carried out parallel. It is evident because before parameter \( d_3 \) stands sign "\( \mp \)"

Consider the fronts of computations for \( Y \) and \( Z \) variables:

\[
\begin{align*}
Y: \quad i & = K_1, \quad -2k + 1 = K_2, \quad -j = K_3 \\
Z: \quad i & = K_1, \quad -2k = K_2, \quad j = K_3
\end{align*}
\]

As the front of computations is set by three equations then both these variables are calculated strictly sequential. Unit matrix of transformation and equations (25) has the following form:

\[
\begin{align*}
Y: \quad -2\bar{k} + 1 & = K_2, \quad \bar{j} = K_3 \\
Z: \quad -2\bar{k} & = K_2, \quad \bar{j} = K_3
\end{align*}
\]

Using the algorithm of reduction we obtain the following fronts of computations:

\[
\begin{align*}
Y: \quad -\bar{k} & = d_2, \quad -\bar{j} = K_3 \\
Z: \quad -\bar{k} & = d_2, \quad \bar{j} = K_3
\end{align*}
\]

Parameter \( t_2 \) has the following values:

\[ t_2^x = 1, \quad t_2^z = 0 \]

Matrix of transformation for obtained fronts of computations has the following form:

\[ A_2 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \]

Transform the coordinates:

\[
\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \bar{j} \\ \bar{k} \end{pmatrix} = \begin{pmatrix} \bar{j} \\ \bar{k} \end{pmatrix}
\]

One equation for the front of computations of \( Y \) variable is left as a result:

\[ -\bar{k} = K_3 \]

and one equation for the front of computation of \( Z \) value:

\[ \bar{k} = K_3 \]

Then we obtain the solution in the initial notations. At first we carry out the transformation of the following type:
\[
\begin{pmatrix}
  j \\
  k
\end{pmatrix} =
\begin{pmatrix}
  0 & 1 \\
  -1 & 0
\end{pmatrix}
\begin{pmatrix}
  a_1 \\
  -a_3
\end{pmatrix} =
\begin{pmatrix}
  -d_1 \\
  -d_2
\end{pmatrix}
\]

And finally we obtain:
\[
\begin{pmatrix}
  i \\
  j \\
  k
\end{pmatrix} = A_3 \cdot
\begin{pmatrix}
  d_1 \\
  -d_3 \\
  -d_2
\end{pmatrix}
\]

The solution for \( Z \) variable is done in the same way.

The scheme of the program is given below.

```
DO 1    d_1 = 1,11
        IF   d_1 \in (1,10) THEN
          C   Z
          C   Z
          C   Y  r_2^y = 1
          DO 2    d_2 = -101
          DO 3    d_3 = 1,10
          3   Z(d_1, d_2, d_3) = f_Z(Y(d_1, d_2, d_3) + 1, Z(d_1, d_2, d_3) - 1)
          C   Y
          DO 4    d_4 = -10 - 1
            Y(d_1, d_2, d_3) = f_X(X(d_1 - 1, d_2, d_3), Y(d_1, d_2 + 1, d_3), U(d_1 - 1, d_2, d_3), Z(d_1, d_2, d_3))
          4 CONTINUE
          2 CONTINUE
          C   V
            IF   d_4 \in (211) THEN
              DO 5    d_5 = 1,10
              C   d_5
              DO 5    d_5 = 1,10
              5   V(d_1 - 1, d_2, d_3) = f_V(U(d_1 - 1, d_2 + 1, d_3), V(d_1 - 1, d_2, d_3 - 1))
              C   X
                IF   d_5 \in (1,10) THEN
                  DO 6    d_6 = 1,10
                  C   d_6
```

\begin{verbatim}
DO 6 d_1 = 1,10
6  X(d_1,d_2,d_3) = f_x(X(d_1,d_2-1,d_3), Y(d_1,d_2,d_3))
C  U
   IF  d_1 \in \{1,10\} THEN
   DO 7 d_1 = -10-1
   C d_1
   DO 7 d_1 = -10-1
      U(d_1,-d_2,-d_3) = f_y(Y(d_1,-d_2,-d_3+1), U(d_1,-d_2,-d_3+1), V(d_1-1,-d_2-1,-d_3))
   7 CONTINUE
1 CONTINUE
\end{verbatim}

PostScript

Special thanks to I.B. Zadykhailo and K.N. Efimkin for their attention to the work and useful notes.