Interval Computations
No 1(3), 1992

CLASSIFICATION APPROACH TO PROGRAMMING OF LOCALIZATIONAL (INTERVAL) COMPUTATIONS

Alexander G. Yakovlev

We propose new language constructs in a Pascal-like language that permit increased expressiveness and effectiveness of programs describing localizational (interval) computations. The constructs generalize traditional language tools such as enumeration type and **case**-construction. Use of the new language tools is founded on a preliminary classification of situations that appear during program execution.

КЛАССИФИКАЦИОННЫЙ ПОДХОД К ПРОГРАММИРОВАНИЮ ЛОКАЛИЗАЦИОННЫХ (ИНТЕРВАЛЬНЫХ) ВЫЧИСЛЕНИЙ

А.Г. Яковлев

Предложены новые языковые конструкции, позволяющие повысить выразительность и эффективность программ, описывающих локализационные (интервальные) вычисления на паскалеподобном языке. Конструкции обобщают такие традиционные языковые средства как перечислимый тип и **case**-конструкция. Применение новых языковых средств основано на предварительной классификации ситуаций, возникающих во время выполнения программы.

Introduction

A considerable part of the programmer's activity deals with classifying situations arising during program execution and directing program behavior in each case. For example, if a Boolean variable L is set through a variable M, then the following two fragments

```
1)$L := M$;
  **if** $L = TRUE$ **then** $i := i + 1$
  **else** $i := i - 1$;
2)$L := M$;
  **case** $L$ **of**
        $TRUE:$  $i := i + 1$;
        $FALSE:$ $i := i - 1$
  **end**;
```

describe two classes of situations ($L = TRUE$ and $L = FALSE$) and indicate what to do when there is a hit in a class.

In both fragments, one may distinguish two kinds of operations: operations to identify a situation (checking what class it belongs to) and operations to enumerate the actions to be performed once a situation has been identified. An important distinction between these fragments is the "degree of procedureness" of the language tools used (the **case**-construction hides the order of the identifying actions). However, a general programming rule of thumb is to use non procedural constructs as much as possible, due to their higher expressive power and due to technology factors; we then resort to procedural constructs only to improve efficiency. That is, if the estimated efficiency is equal or higher, it is preferable to use non procedural means. In particular, the **case**-construction is preferable to the **if**-construction for binding of identified situations with their corresponding actions. (Also, the **case**-construction's reliability is higher, too.)

However, the traditional **case**-construction has an essential drawback: it presupposes that the situations are mutually exclusive. In practice, it often occurs that some situations are special cases of others. Also, classes of situations may thus intersect and, in particular, be embedded in each other. Such a structure of situations is characteristic for classical interval computations [1–3], and, in a more general setting – localizational ones [4]. Note that a localizational computation is a process of constructing a set (called a shell) to which an unknown mathematical object (called a kernel) belongs; usually the object is a solution of some numerical problem in classical mathematics. In [4] a pair consisting of a kernel and its shell is called a locus. The simplest example of a locus is an interval: "An interval has a dual nature as both a *number* and a *set* of "real" numbers. Many of the algorithms for interval methods make use of this duality and combine set theoretic operations such as set intersection with arithmetic operations" [1]. In other words, from the point of view of localizational computations, an interval is a pair (x, X), where x is an unknown real number, such that $x \in X = \{x | x_1 \leq x \leq x_2\}$. In the discussion to follow, it is sufficient to view a locus to as such an interval.

If one considers the possible relationships between two intervals as classes of situations, then it is easy to see that many of them intersect. Because correlation of locuses is one of the most used operations in localizational computations, one needs adequate language constructs to clearly and easily describe branching of localizational programs.

The first attempt to achieve this was undertaken by E.A.Musaev [5,6], who proposed **if**-construction based on three valued logic ("yes", "no", "unknown") of conditional expressions. (Evidently, the possibility of using three valued logic for this purpose was first pointed out by R.E.Moore as early as in 1968 [7].)

More universal language tools were proposed in [8]. This paper is a refinement of that paper. These tools were developed as a strict extension of Pascal-like languages because only Pascal-like languages contain sophisticated data type mechanisms necessary to achieve the stated goals. Moreover, they are widely used in practical programming.

The developed language tools allow easy handling of non-intersecting classes of situations and combine the advantages of **case**-construction with the efficiency of **if**-construction. Also, these tools may be used outside the context of localizational computations.

1. A collection of relations

As stated above, intervals act like sets in set theoretical operations and like numbers in arithmetical operations. Usually, the symbol associated with an operation indicates whether it deals with numbers or sets. However, a difficulty arises with the comparison operations: in both mathematical notation and in a programming language, one should use separate signs for relations for intervals as sets and for intervals as numbers.

Let us clarify the point using the relation "$\leq$" as an example. Let A and B be arbitrary intervals (closed, open, semi-open or infinite). Then the relation

$$A \leq B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a \leq b) \tag{1}$$

binds A and B as real numbers because the relation of the same name binds each pair of real numbers lying within these intervals. But if one defines "$\leq$" as

$$A \leq B \equiv \underset{a'\in A}{(\forall a')}\underset{b'\in B}{(\exists b')}\ (a' \leq b')\ \&\ \underset{b''\in B}{(\forall b'')}\underset{a''\in A}{(\exists a'')}\ (a'' \leq b'') \tag{2}$$

then, obviously, here set shells are bound, since the relations of the same name bind corresponding bounds of intervals.

Both definitions have equal rights for existence and are used in appropriate contexts. So, e.g., if one checks whether a quadratic equation has complex roots, then the relation "$\leq$" is understood in the sense (1) in the expression $4ac \leq b^2$. However, if one checks whether one should continue iterations for computing a constant-sign series with positive terms, then to compare partial sums the relation $S_n \leq S_{n+1}$ is understood in the sense of (2).

Therefore, one should put two relations in correspondence with each of the symbols "$<$", "$\leq$", "$=$", "$\geq$", "$>$" and "$\neq$". However, these 12 generated relations are still not enough. Since intervals act either as numbers or as sets, one needs classical relations such as "$\subset$", "$\subseteq$", "$\supset$" and "$\supseteq$" to work with the sets. Moreover, it may prove useful to have relations "*to contain more than one common point*", "*to have the same right (left) bounds*", etc.

Unfortunately, authors of most theoretical works on interval mathematics introduce only a small subset of the relations of practical significance, and do not mention the need a fully versatile set in applications. So, e.g., in [3] only the relation "$=$" for intervals as sets has been defined; in [2] – the relation (2) and "$\subset$", etc. However, practical needs demand enrichment of a language to program localizational algorithms, and also require a set of primitives used to translate such a language with a larger collection of relations. We review the sizes of such collections in some languages described in literature (relations between elements of types ***real*** and *interval* are considered along with relations between elements of type *interval*): Triplex-Algol [9,10] – 2 relations, Algol-68 with interval data types [11,12] – 6 (the authors mention the advisability of enriching this collection), Fortran-SC (now it has renamed to ACRITH-XSC) [13,13a] and Pascal-SC (-XSC) [14,14a] – 8.

Also, 15 relations have been implemented in one of the most recognized packages for interval operations [15]; the exact same 15 relations have also been implemented in [16].

It is advisable to designate each relation in the language by a special symbol. Let us review possible designations and corresponding definitions for some relations. (This notation will be used in the subsequent examples).

$$A < B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a < b)$$

$$A > B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a > b)$$

$$A \neq B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a \neq b)$$

$$A = B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a = b)$$

$$A \leq B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a \leq b)$$

$$A \geq B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a \geq b)$$

$$A \geq B \equiv \underset{a\in A}{(\forall a)}\underset{b\in B}{(\forall b)}\ (a \geq b)$$

$$A \approx B \equiv \underset{a'\in A}{(\forall a')}\underset{b'\in B}{(\exists b')}\ (a' = b')\ \&\ \underset{b''\in B}{(\forall b'')}\underset{a''\in A}{(\exists a'')}\ (a'' = b'')$$

$$A \lesssim B \equiv \underset{a'\in A}{(\forall a')}\underset{b'\in B}{(\exists b')}\ (a' \leq b')\ \&\ \underset{b''\in B}{(\forall b'')}\underset{a''\in A}{(\exists a'')}\ (a'' \leq b'')$$

$$A \gtrsim B \equiv \underset{a'\in A}{(\forall a')}\underset{b'\in B}{(\exists b')}\ (a' \geq b')\ \&\ \underset{b''\in B}{(\forall b'')}\underset{a''\in A}{(\exists a'')}\ (a'' \geq b'')$$

$$A \supset B \equiv \underset{b\in B}{(\forall b)}\underset{a'\in A}{(\exists a')}\underset{a''\in A}{(\exists a'')}\ (a' < b < a'')$$

$$A \subset B \equiv \underset{a\in A}{(\forall a)}\underset{b'\in B}{(\exists b')}\underset{b''\in B}{(\exists b'')}\ (b' < a < b'')$$

The symbol "$\leq$" is now understood in the sense of (1).

Relations in the "interval" language have a rather complex structure, which will now be considered.

2. Taxonomic structure of relations and generalized enumeration type

Let R_ϕ denote the set of ordered pairs of intervals binded by the relation ϕ. It is easy to see that $R_< \subset R_\leq \subset R_\lesssim$, $R_> \subset R_\neq$, etc. The strict inclusion relation generates some structure on the set of all R_ϕ. If the following condition holds

$$(\forall\alpha)(\forall\beta)(\exists\gamma)(R_\alpha \cap R_\beta \neq \varnothing \Rightarrow R_\alpha \cap R_\beta = R_\gamma),$$

then such a structure is called taxonomic, and sets R_ϕ are called taxons. Various types of taxonomic structures have been described in [17].

Every taxonomic structure may be represented as a connected acyclic oriented graph. In such a graph, a vertex corresponds to a taxon and an edge corresponds to an inclusion. The relations defined in Sec. 1 form the structure shown in Fig. 1.

Note that the root vertex R_0 corresponds to the maximal taxon (the taxonomic universe), i.e. just the set of all ordered pairs of intervals.

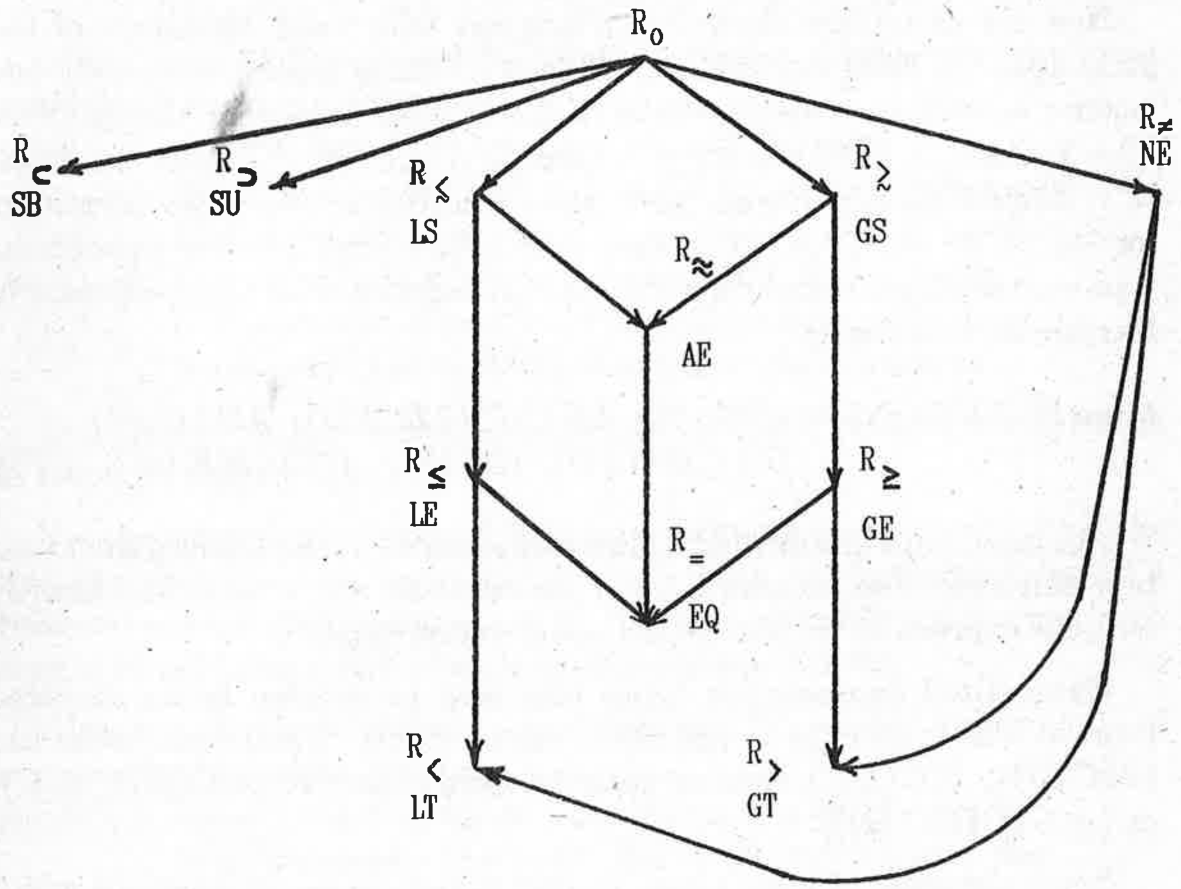

Fig.1

In what follows, we also need another representation of a taxonomic structure, a linear parenthesis one, constructed by the following rules: following the symbol for the relation corresponding to some vertex in the graph, we list in parentheses (and separate by commas) the symbols corresponding to the vertices on edges of the graph which begin at the given vertex; enumerating starts from the root, denoted by $\circ$. To make this mapping of the graph into the parenthesis structure single valued, an order for listing the vertices within a given parentheses level should be specified. In our case, a linear parenthesis representation may look, for example, like

$$\circ(\subset,\ \supset,\ \lesssim\ (\leq\ (<,\ =),\ \approx\ (=)),\ \gtrsim\ (\approx\ (=),\ \geq\ (=,\ >)),\ \neq\ (>,\ <)).$$

The first symbol in such an expression will be omitted later.

Now let us replace the relation symbols with valid identifiers of the language. We shall suppose that these identifiers consist of exactly two letters; we will use capital letters in the English names for the symbols: $\subset$ – SuBset, $\supset$ – SUperset, $\lesssim$ – Less as a Set, $\gtrsim$ – Greater as a Set, $\approx$ – Approximately Equal, and, also, traditional two-letter identifiers for $<$, $>$, $=$, $\neq$, $\leq$, $\geq$. It is then easy to see that a linear parenthesis representation generalizes a notation of an enumeration type, common in Pascal-like languages:

$$\textbf{type}\ \text{RELATIONS} = (SB,\ SU,\ LS\ (LE\ (LT,\ EQ),\ AE\ (EQ)),\\ GS\ (AE\ (EQ),\ GE\ (EQ,\ GT)), NE\ (GT,\ LT)).$$

Actually, for a graph with a diameter equal to 1, the linear parenthesis representation does not have inner parentheses, and it coincides exactly with an expression for a common enumeration type.

Generalized enumeration types may also be written in an abridged form in which no edge is described twice. If, say, a full form looks like $(A(C(D)),\ B(C(D)))$, then an abridged form looks like $(A(C(D)),\ B(C))$ or $(A(C),\ B(C(D)))$.

Some elements of a type may receive synonymous designations which are equivalent in all possible contexts. Synonymy is given in a **type** description: synonymous names are separated by the equality sign. A clever use of synonyms may enhance the expressiveness of program texts, especially if in the construction of identifier symbols from different alphabets and special symbols are used.

This is a variant of a statement of the type RELATIONS in the abridged form with synonymous designations of the vertex SB:

$$\textbf{type}\ \text{RELATIONS} = (SB = SuBset = \subset, SU, LS(LE(LT, EQ),\\ AE(EQ)), GS(AE, GE(EQ, GT)), NE(GT, LT)).$$

Elements of a generalized enumeration type, like elements of a common enumeration type, may be arguments and values of functions, they may be combined into sets, etc. Accordingly, predefined standard functions such as finding a predecessor and a successor of an element (*pred* and *succ* in the Pascal notation) are generalized. The arities of some of them then increase. For example, it is convenient to define *pred* and *succ* in a general case as two-placed functions (a, b and c belong to the same generalized enumeration type):

$pred(a, b)$ takes a value c where c is a name of a vertex which is a predecessor of b in the list of all vertices such that there are edges from the vertex a to these vertices;

$succ(a, b)$ takes a value c where c is a name of a vertex which is a successor of b in the list of all vertices such that there are edges from the vertex a to these vertices.

Ordering of the list corresponds to ordering of the vertices enumeration in the type description.

For the generalized standard functions *pred* and *succ* one may omit the first operand: the root vertex is assumed. Thus, in the special case of a common enumeration type, a call to the generalized functions *pred* and *succ* is identical to a call to their traditional analogues.

Also, specific functions and operations may be applied to elements of a generalized enumeration type, e.g.:

$pred2(a, b)$ takes a value c, where c is a name of a vertex which is a predecessor of b in the list of all vertices such that there are edges from these vertices to the vertex a;

$succ2(a, b)$ takes a value c, where c is a name of a vertex which is a successor of b in the list of all vertices such that there are edges from these vertices to the vertex a;

a **in** b takes a value $TRUE$ if there is a way from b to a, otherwise $FALSE$.

The functions *pred*, *succ*, *pred*2 and *succ*2 form a basis for description of all actions which are necessary to deal with a generalized enumeration type. One may recommend that the operation **in** be predefined because its use is expected to be rather frequent, and it can be implemented efficiently in a "built in" language variant.

To make this clearer, we give examples of evaluation of the proposed functions for the type RELATIONS described above:

$$pred\,(LE,\ EQ) \;=\; LT;$$
$$succ\,(GS) \;=\; NE;$$
$$pred2\,(EQ,\ succ\,(GS,\ AE)) \;=\; AE.$$

3. Derived generalized enumeration types

One may construct derived generalized enumeration types based on the basic generalized enumeration type. They are generated by only two operations on a graph corresponding to a basic type:

1) deleting explicitly pointed edges along with those vertices and edges such that the path from the root to them consists of only these explicitly pointed edges;
2) joining of vertices, unconnected in the initial graph, or of explicitly pointed vertices with new subgraphs.

The first operation is denoted by the key word **delete** and the second one – by **join**. An enumeration of the edges to be deleted, separated by commas follows the word **delete**. An element of the list may have one of the following three forms:

(X, Y)	– an edge from a vertex X to a vertex Y,
$(X, *)$	– all edges going from a vertex X,
$(*, Y)$	– all edges to a vertex Y.

Let us consider an example for the type RELATIONS: the operation **delete** (LS, LE), $(*, EQ)$, $(GE, *)$ deletes edges (LS, LE), (LE, LT), (LE, EQ), (AE, EQ), (GE, EQ), (GE, GT).

Similarly, a list whose elements are separated by commas follows the key word **join**. An element of the list is a subgraph written in the syntax of a linear parenthesis structure. Identifiers from both the list following the word **join** and from the description of a basic type name vertices to which the join operation applies.

Suppose, for example, that the type RELATIONS is transformed in such a way that the vertex LS depends on SU, and SB is joined with a new vertex A and, and a new vertex B depends on A. This would be written as follows: **join** $SU(LS)$, $SB(A(B))$. Thus, vertices of the basic type may be connected and new vertices may be joined to new vertices.

Let us consider a realistic example to demonstrate use of a derived generalized enumeration type.

Suppose that, instead of the type RELATIONS, we would like to use a type NEW_RELATIONS containing, instead of the relation $\subset$, a relation $\subseteq$, defined in the following way:

$$A \subseteq B \equiv \underset{a\in A}{(\forall a)}\underset{b'\in B}{(\exists b')}\underset{b''\in B}{(\exists b'')}\ (b' \leq a \leq b'').$$

It is not difficult to see that $R_{\subseteq}$ intersects partially with $R_{\lesssim}$, $R_{\gtrsim}$, $R_{\leq}$, $R_{\geq}$ and strictly contains $R_{\approx}$ and $R_{=}$. Some of newly created taxons also intersect with one another. The resulting taxonomic structure may be described by the following data type (the relation $\subseteq$ is put into correspondence with the element $SB2$; to make the new names more obvious, names of new vertices are constructed by concatenating the names of vertices upon which they depend):

type NEW_RELATIONS $= (SB2\,(SB2LS\,(SB2LSLE), AE, EQ,$
$SB2GS, (SB2GSGE), SU, LS\,(SB2LS, LE$
$(SB2LSLE, LT, EQ), AE\,(EQ)), GS\,(SB2GS,$
$AE, GE, (SB2GSGE, EQ, GT)), NE\,(GT, LT)).$

Although the description of the type NEW_RELATIONS is given in an abridged form, it looks cumbersome. It may be made more compact if the type NEW_RELATIONS is described as a derived one from RELATIONS. For this purpose let us use a key word **derfrom** ("derived from") indicating the type from which the given type is derived:

type NEW_RELATIONS = **derfrom** RELATIONS **delete** $(*, SB)$
join $SB2\,(SB2LS\,(SB2LSLE), AE, EQ,$
$SB2GS\,(SB2GSGE), LS\,(SB2LS), LE$
$(SB2LSLE), GS\,(SB2GS), GE\,(SB2GSGE)).$

Note that both the basic and derived generalized enumeration types are static – execution of the operations **delete** and **join** takes place at compile time. In the description of a derived type, an arbitrary number of the operations **delete** and **join** may occur; these operations are executed strictly in the order of their occurrence.

The possibility of describing new types as derived from already available ones may be of great practical importance for languages, allowing export and import of types. For an already available module (package) within which a large and complex taxonomic structure is defined, modifying this structure to meet certain requirements may be much easier than constructing a new structure from scratch. This is clearly demonstrated by the above example of a taxonomic structure of relations between intervals.

To conclude this section, let us indicate that the idea of using derived types goes back to the language Ada (the Ada counterpart of the keyword **derfrom** is the keyword **new**, less understandable in this context). However, a derived type in Ada is constructed only by limiting a basic (parental) type, while, in our case, extending, as well as limiting, is possible. Let us also note that a possible (in Ada, too) collision between elements with the same names from different enumeration types can be settled by giving explicit qualifiers for the corresponding types.

4. Correlation function and its realization

A correlation function is a function of two interval arguments whose value is an element of a generalized enumeration type. The value denotes a relation which binds the arguments of the function. A value of the function corresponds to a taxon in the taxonomic structure of relations between pairs of intervals. It is natural to choose the minimal such taxon. Thus, for example, for the pair $([2;3],[7;10])$, the relations $\lesssim$, $\leq$, $\neq$, $<$ hold, but the most "precise" characterization of this pair is given by the relation "<". It is thus advisable to set $rel\,([2;3],[7,10]) = LT$ where rel is the name of the correlation function. (This notation will be used later, too.) In addition to maximal "precision", it is natural to demand yet another property from the correlation function – that it be totally defined (i.e. there is no pair of intervals for which it would be impossible to set, by a correlation function, a relation which binds them). If necessary, it is easy to make a correlation function totally defined in a somewhat artificial way: by adding a special taxon to the taxonomic structure to complete the taxonomic universe.

The purpose of a correlation function is, obviously, an automated classification or, in other words, identification of situations.

Let us study implementation of correlation functions.

To discuss this question, non constructive definitions of relations (like those given in Section 1) are not enough. Rules of identification (constructive definitions) will be introduced through relations between bounds of intervals. For the sake of simplicity we shell limit ourselves to closed intervals and to the collection of relations considered in Section 1. Decision tables with bounded exit [18] will be used for the notation.

Such a table is shown in Fig.2. As is customary, it is divided into two parts. The symbol "+" in the upper part of the table signifies fulfillment of a condition from a corresponding line, while a blank position means a neutral state. The structure of the lower part of the table is standard.

$a_1 = a_2$										+	
$b_1 = b_2$											+
$a_1 < b_1$		+	+								
$a_1 = b_1$									+	+	+
$a_1 > b_1$	+					+					
$a_2 < b_1$					+						
$a_2 = b_1$				+							
$a_2 > b_1$			+								
$a_1 < b_2$						+					
$a_1 = b_2$							+				
$a_1 > b_2$								+			
$a_2 < b_2$	+		+								
$a_2 = b_2$									+	+	+
$a_2 > b_2$		+				+					
$A \subset B$	+										
$A \supset B$		+									
$A \lesssim B$			+	+	+				+	+	+
$A \leq B$				+	+					+	+
$A < B$					+						
$A \gtrsim B$						+	+	+	+	+	+
$A \geq B$							+	+		+	+
$A > B$								+			
$A \approx B$									+	+	+
$A = B$										+	+
$A \neq B$					+			+			

Fig.2

The main problem associated with realization of a correlation function consists of considering the ways of obtaining a procedural representation for a decision table. An efficient realization of such a mapping (i.e. of a correlation function) is not as simple a task as it may appear at the first sight. The point is that one should consider such factors as the computational complexity of the testing conditions from the upper part of the table and the probabilities of the identifications from its lower part. The complexity of construction of an optimal procedural representation grows combinatorially as a function of the number of conditions. Therefore, standard methods of obtaining optimal or sub optimal procedural representations as well as every possible trick to improve the final result are of interest [18]. Striving for efficiency is reasonable here because a correlation function will almost always be computed within some loop, and an efficiency difference of several orders of magnitude is possible in different implementations.

The information which may help increase the efficiency of an implementation of a correlation function is divided into a priori (known before a program execution) and a posteriori (appearing during a program execution) information. To obtain the information, we must analyze the expected dependencies (including static ones) in the input data, as well as particular features of the algorithms in which correlation of intervals will be tested. It may then turn out that requirements on a correlation function will be substantially different for different algorithms or for different parts of the same algorithm. In this case it is advisable to use procedures, prepared beforehand, whose calls are semantically equivalent but are implemented in different ways. Note that a part of work in forming the text of these procedures, including their specialization, may be given to a preprocessor, if a decision table is kept in the form of a parametrized macro (like a *generic component* in the language Ada). Parameter values may be defined by the results of an automated analysis of the context in which a correlation function is called.

A posteriori information may be considered by reordering testing of conditions during program execution. Whether reordering is carried out or not depends on statistical properties of a function of pairs of intervals present upon entry each time a function is called. For a discussion of such reordering methods, see [19] Section 6.1 "Consecutive Search", Subsection ⟪"Self-organizing" File⟫ and, also, the exercises (and the corresponding answers to them) for this subsection.

It is interesting to note that, the more natural a classification reflected by a taxonomic structure is, the more closely the rank distribution of the number of hits of each of the taxons resembles the hyperbolic distribution (the Zipf's distribution). This regularity, empirically verified repeatedly, was given a theoretical ground in [17]. In this light, tuning of the algorithm for computing a correlation function to some concrete context of its use seems somewhat important.

The advisability of special design of modules (packages) containing descriptions of complex taxonomic structures was stressed at the end of the previous section. An integral part of such modules (packages) should consist of procedures implementing corresponding correlation functions in different ways (depending on potential conditions of use). The modules, tuned in such a way, substantially enhance development of automated systems for interval computations [20].

5. Organization of branching and generalized case-construction

The simplest way to organize the branching in interval programs consists of using the traditional **if**-construction where conditional expressions contain operations to correlating intervals:

$$\textbf{if } A \lesssim B \textbf{ then} \dots$$

This way is indispensable when it is necessary to check whether a pair (A, B) belongs to a concrete relation. Perhaps the only difficulty encountered while organizing the branching in this way is a lack of necessary predefined relations and an impossibility to define them. In principle, it is easy to overcome the difficulty with the use of two placed logical functions:

$$\textbf{if } ls(A, B) = TRUE \textbf{ then} \dots$$

However, such a notation makes program texts more cumbersome and less expressive. Besides, one needs a complete library of logical functions. But such a library can be dispensed with by using a generalized enumeration type, a correlation function and the operation **in**:

$$\textbf{if } rel(A, B) \textbf{ in } LS \textbf{ then} \dots$$

Clearly, if rel is not tuned to be used in the concrete context during the process of translation of the body of the procedure (as was said before), then the speed of checking the corresponding relation may prove lower than in the previous cases.

Thus, the simplest way to simultaneously achieve expressiveness and efficiency is to provide the capability to define new relations in the language. Such capabilities are present, for example, in universal languages Algol-68 and Ada (but lacking in Pascal and Modula-2), and in contemporary SC-languages [13a,14a].

So far, we have dealt with organizing the binding of only one situation with corresponding actions. However, when comparing intervals in interval programs, one should expect, as a rule, a detailed analysis of possible ways that two intervals can be related, along with the actions to be undertaken in each of the cases. The tool for checking the relationship between two intervals (identification) has been already presented – it is a correlation function. The remaining task is to connect a taxonomic structure of situations (relations) with the imperative structure of a **program** (a description of actions to undertake). The generalized **case**-construction serves this purpose.

Syntactically, the generalized **case**-construction differs from the traditional one only in one way: its selector and labels of variants belong to the (same) generalized enumeration type.

Performing a **case**-statement generated by the **case**-construction proceeds in the following way: the branches are chosen whose labels name the vertices lying on all the routes from the root of the graph, corresponding to the selector type, to the vertex whose name is the selector value. Three cases may then occur:

1) only one label satisfies the condition given; in this case the control is just transferred to the corresponding branch;
2) there are no such labels; one then performs the **else**-branch, if any, and, if not, the program terminates with a failure;
3) there are several such labels; if each of them belongs to the list of labels of the same variant, then the branching just follows this variant. If they belong to different variants, then one searches for all minimal taxons from taxons corresponding to the selected labels. The branching follows the last variant from the set of variants marked with labels corresponding to these minimal taxons.

It is not difficult to see that if a selector of a generalized **case**-construction belongs to a standard enumeration type, then the generalized **case**-construction also reduces to a standard one.

Execution of a **case**-statement may be illustrated with the following example:

```
case rel(A, B) of
  LS, LE : ... |
      GE : ...
  else
end,
```

where $rel(A,B)$ takes values of the type RELATIONS, and the sign "|" is a delimiter of a variant (in the Modula-2 notation). The first variant will be chosen for, e.g., the following pairs (A, B): ([1;5], [2;7]), ([1;5], [5;7]), ([1;5], [1;5]), ([1;5], [6;7]); the second one – for ([5;7], [1;5]), ([6;7], [1;5]), ([1;1], [1;1]); the **else**-branch – for ([1;7], [2;5]), ([2;5], [1;7]), etc.

A statement which is semantically equivalent to the above fragment might be written in the terms of an **if**-construction with **elsif**-branches. On the surface, this way seems to be even more natural. However, the approach based on the generalized **case**-construction gains expressiveness as well as, in general, efficiency. The gain in expressiveness is due to the fact that all the variants are explicitly named, and these names are separated from the actual imperative part of the statement. The gain in efficiency is due to the fact that consecutive identification of situations may lead to the need to repeatedly check the same elementary conditions and exclude the possibility of "self-organization" mentioned in Section 4. For example, for independent identification of LS and LE, one would need to check the conditions $a_1 = b_1$ and $a_2 = b_2$ twice, as is clearly seen in Fig.2.

Now let us present yet another possible improvement of the traditional **case**-construction.

Introduction of this improvement is motivated by the following consideration: since representatives of each subtaxon inherit features of the parental taxon, one would expect the processing of representatives of these taxons to include (at least partially) the same operations. Therefore, it is advisable to have a construction allowing exclusion or, at least, reduction of duplication of coinciding operations on different branches of the **case**-statement. Certainly, duplication may be avoided in a traditional way by placing coinciding statement sequences into separate subprograms. However, in our case it can be done with a special construction **like** which is substantially more explicit in this context and has a more efficient implementation.

The construction **like** may occur in every place of a variant of a **case**-

statement. It creates the statement **like** whose only argument is a label of one of variants of the **case**-statement containing the given **like**-statement. The execution of a **like**-statement consists of an unconditional transfer of control to the beginning of the variant with the label indicated, and a subsequent return to the point of call (i.e., it resembles an execution of a subprogram without parameters). The construction **like** also may be viewed as a macro whose processing results in substituting a **like**-statement for the body of the corresponding branch.

Let us consider an example illustrating use of the **like**-construction. Suppose we have the following **case**-statement:

```
case $S$ of
    $M1:\ B := B + 1;\ C := C + 1\ |$
    $M2:\ A := A + 1;\ B := B + 1;\ C := C + 1;\ D := D + 1\ |$
    $M3:\ A := A + 1$
end.
```

A semantically equivalent fragment may be written with the help of the **like**-construction:

```
case $S$ of
    $M1:\ B := B + 1;\ C := C + 1\ |$
    $M2:$ like $M3$; like $M1$; $D := D + 1\ |$
    $M3:\ A := A + 1$
end.
```

If several **like**-statements are components of a single **case**-statement, then it is natural to require a sequence of macro generations to be finite. Control of checking of this requirement may be given to the compiler.

If one represents the statements constituting the parts of the **case**-statement with **like**-statements in branches, as vertices and potential transfers of control between them as edges, then the resulting graph will be connected, acyclic and oriented. Thus, the **like**-construction facilitates establishing a connection between a taxonomic structure of situations and an imperative structure of variants.

To conclude this section, we must make the following remark. If a language allows composition of identifiers of symbols from a sufficiently representative set, and if visualization devices can display these symbols, then specially designed pictograms may be used to denote elements from an enumeration type. This will undoubtedly enhance the expressiveness of program texts. In the example in Section 6, elements of the type RELATIONS are assumed to be directly represented by the symbols of the corresponding relations.

6. An example of use of the proposed language tools

Let us demonstrate possibilities of using of the language means proposed above with the following simple example.

Consider the task of finding real roots of a function $f(x) = ax^2+bx+c$, where a, b and c are floating point numbers. Since the discriminant of a quadratic equation does not have an exact computer representation for every a, b and c, it is natural to do the task localizationally, i.e. not to seek nuclear real roots, but to seek their interval shells.

During the solution process, it becomes necessary to distinguish five cases of the position of the interval $D = [d_1, d_2]$, containing the exact value of the discriminant, with respect to zero (the interval $[0, 0]$):

1) $d_1 \geq 0,\ d_2 > 0$;
2) $d_1 = d_2 = 0$;
3) $d_2 < 0$;
4) $d_1 < 0,\ d_2 = 0$;
5) $d_1 < 0,\ d_2 > 0$.

In cases (1) and (2), roots are found by the standard rules

$$x_{1,2} = (-b \pm \sqrt{D})/2a \qquad \text{and} \qquad x_1 = x_2 = -b/2a,$$

respectively. (All operations are executed as interval-arithmetical ones). Case (3) does not cause problems – obviously, real roots do not exist. Cases (4) and (5) are somewhat harder. In (4), the correct solution looks like "Real roots are absent or $x_1 = x_2 = -b/2a$". Thus, (4) actually breaks down into (2) and (3). Similarly, (5) breaks down into (1) and (3), where the interval $[0, d_2]$ is taken as D for calculation of x_1 and x_2 in case (1).

The proposed constructions and a sufficiently rich collection of predefined relations allow description of the solution of the given task in terms of operations on objects of an abstract interval data type. In the program, the following functions are used: *rel* (see Section 2) with values of the type RELATIONS, *left* and *right* (whose argument is an interval and whose value is a degenerate interval, numerically equal to the left (right) bound of the argument), *compose* (whose arguments are a two degenerate intervals and whose value is an interval composed of arguments taken as bounds). We suppose that all variables are of the scalar interval data type and both the arithmetical operations and the operation of extracting a square root are performed as interval operations. The accuracy of computations (the width of the resulting intervals) depends only on the number of digits used in intermediate results.

```
$D := b**2 - 4*a*c$;
case $rel(D,\ [0,0])$ of
  $\geq$ : $x1 := (-b + \mathrm{sqrt}(D))/(2*a)$;
       $x2 := (-b - \mathrm{sqrt}(D))/(2*a)$;
       $print('x1 =',\ x1,\ ',\ x2 =',\ x2)$ |
  $=$ : $x1 := -b/2*a$;
       $print('x1 = x2 =',\ x1)$ |
  $<$ : $print($'Real roots are absent'$)$ |
  $\leq$ : like $<$; $print($' or '$)$; like $=$ |
  $\supset$ : like $<$; $print($' or '$)$;
       $D := compose(left([0,0]),\ right(D))$;
       like $\geq$
 end.
```

In accordance with the remarks in Section 4, the function *rel* may be specialized for identification of only the five relations actually used. "Self-organizing" may prove to be useful, too, since, obviously, in actual applications of this program, the different branches will be selected with frequencies, far from being equal.

As we see, the proposed language tools allow the writing of programs for localizational computations in a compact and expressive way with subsequent generation of an efficient object code.

Conclusion

In the present paper, some new constructions are proposed which enhance possibilities of Pascal-like languages. Use of these constructions is illustrated in examples concerning the organization of branching in localizational programs. All the given examples deal with correlation of standard closed intervals only. At the same time, modern interval analysis uses various generalizations of closed intervals: irregular intervals, multi-intervals, etc. [21]. Their correlation generates new relations, stressing the necessity of special language tools to deal comfortably and efficiently with them.

One should note that complex structures of relations do not arise only in localizational programs. Thus, for example, during 1985-1987 in the USA, national (actually, international) standards for arithmetic for floating point numbers have been adopted [22–23]. According to these standards, 26 types of relations are supposed to hold between objects written in the format of floating point numbers. This generates a sufficiently complex taxonomic structure. Correspondingly, traditional language tools are not enough to deal comfortably with such a structure.

In reality, the proposed language means may also find broad applications beyond numerical programming. The main part of the programmer's activities, irrespective of the application area, consists of classifying situations and defining a program behavior in each of the situations. The generalized enumeration type and the **case**-construction will give the programmer more adequate language tools than the traditional ones.

In the present paper, we have considered use of the generalized enumeration type and the **case**-construction only for dealing with taxonomic structures. Actually, these language means also form a basis for enriching a language with other classification structures – meronomic ones (archetypes) [17]. Thus, elements of the generalized enumeration type may be used for forming the descriptor of variant records. Therefore, the variant record type itself can be generalized. Accordingly, the generalized **case**-construction appears in its definition. Since the record type is intended, first and foremost, for describing a structure of objects, then, from the classification point of view, generalization of this type implies extension of possibilities for constructing archetypes. More detailed consideration of the meronomic aspects goes beyond the scope of the present paper.

The language is known to always influence the programmer's way of thinking. In conjunction with regular use of classification structures in the language, it is valid to speak of forming of the classification style of programming and of appearance of the corresponding programming technology [24,25]. Growing interest in the classification aspects of the programmer's activities would lead to further development of the proposed language tools.

References

1. Moore R.E. *"Methods and applications of interval analysis"*, SIAM Studies in Applied Mathematics. SIAM, Philadelphia, Pennsylvania, 1979.
2. Kalmykov S.A., Shokin Yu.I., Yuldashev Z.Kh. *"Methods of interval analysis"*. Nauka, Novosibirsk, 1986 (in Russian).
3. Alefeld G., Herzberger J. *"Introduction to interval computations"*. Academic Press, New York etc., 1983. (Russian translation: Mir, Moscow, 1987.)
4. Yakovlev A.G. Locuses and localizational computations. *In "Conference on Interval Mathematics, [Saratov], May 23-25, 1989"*, pp. 54-56. Saratov, 1989 (in Russian).
5. Musaev E. Some ways to support interval computations in high-level languages. *In "International workshop on reliability in computing – the role of interval methods in scientific computing, Columbus, Ohio, USA, 8-11 Sept. 1987 : Abstracts of invited lectures and poster talks"*, pp. 19-20. The Ohio State University, 1987.
6. Musaev E.A. The support of interval computations in high-level languages. *"Proc. 1-st Sov.-Bulg. Seminar on Numerical Processing, Pereslavl-Zalessky, Oct. 19-24, 1987"*, Program Systems Institute of the USSR Academy of Sciences, Pereslavl-Zalessky, pp. 110-121 (1989), deposited in VINITI 21.04.89, N 2634-B89 (in Russian).
7. Moore R.E. Practical aspects of interval computation. *Apl. Mat.* **13**, 52-92 (1968).
8. Yakovlev A.G. Organization of branching in interval programs. *"Proc. 1-st Sov.-Bulg. Seminar on Numerical Processing, Pereslavl-Zalessky, Oct. 19-24, 1987"*, Program Systems Institute of the USSR Academy of Sciences, Pereslavl-Zalessky, pp. 147-173 (1989), deposited in VINITI 21.04.89, N 2634-B89 (in Russian).
9. Apostolatos N., Kulisch U., Krawczyk R., Lortz B., Nickel K., Wippermann H.-W. The algorithmic language Triplex-Algol 60. *Numer. Math.* **11**, 175-180 (1968).
10. Cole A.J., Morrison R. Triplex: a system for interval arithmetic. *Software – Pract. Exper.* **12** (4), 341-350 (1982).

11. Guenther G., Marquardt G. A programming system for interval arithmetic in Algol 68. *In "Interval mathematics 1980"* (K.Nickel, ed.), pp. 355-366. Academic Press, New York etc., 1980.

12. Guenther G., Marquardt G. A programming system for interval arithmetic in Algol 68. *Math. Centre Tracts* **134**, 201-215 (1981).

13. Bleher J.H., Rump S.M., Kulisch U., Metzger M., Ullrich Ch., Walter W. FORTRAN-SC. A study of FORTRAN extension for engineering/scientific computation with access to ACRITH. *Computing* **39** (2), 93-110 (1987). – **Reprinted in:** *"Scientific computation with automatic result verification : Papers presented at a conf., Sept. 30 - Oct. 2, 1987, Karlsruhe"* (U.Kulisch and H.J.Stetter, eds), pp. 227-244. Springer (Computing, Suppl. 6), Wien and New York, 1988.

13a. *"ACRITH-XSC: IBM High Accuracy Arithmetic – Extended Scientific Computation. Version 1, Release 1".* IBM, 1990.

14. Bohlender G., Rall L.B., Ullrich Ch., Wolff von Gudenberg J. *"PASCAL-SC: A Computer Language for Scientific Computation".* Academic Press (Perspectives in Computing, vol. 17), Orlando, 1987.

14a. Klatte R., Kulisch U., Neaga M., Ratz D., Ullrich Ch. *"PASCAL-XSC. Language reference with examples".* Springer Verlag, Berlin etc., 1991.

15. Yohe J.M. Software for interval arithmetics: a reasonably portable package. *ACM Trans. Math. Software* **5** (1), 50-63 (1979).

16. Rogalyov A.N., Shokin Yu.I. "A package of interval operations for the BESM-6 computer. Preprint 24-81". Siberian Branch of the USSR Academy of Sciences, Institute of Theoretical and Applied Mechanics , Novosibirsk, 1981 (in Russian).

17. Shreyder Yu. A., Sharov A.A. "Systems and models". Radio i svyaz, Moscow, 1982 (in Russian).

18. Humby E. "Programs from decision tables". MacDonald (Computer monographs), London etc., 1973.

19. Knuth D.E. "The art of computer programming. Vol. 3: Sorting and searching." Addison Wesley (Addison-Wesley series in computer science and information processing), Reading etc., 1973.

20. Yakovlev A.G. What should an automatized system of interval computations be? *In "Inf.-operat. material (interval analysis), Preprint VTs SO AN SSSR, N 6"*, pp. 42-44. Krasnoyarsk, 1988 (in Russian).

21. Yakovlev A.G. Interval computations on electronic computers (a brief survey). Addition to the *Russian translation of "Introduction to interval computations" by G.Alefeld and J.Herzberger,* pp. 336-352. Mir, Moscow, 1987 (in Russian).

22. "An American national standard. IEEE standard for binary floating-point arithmetic : ANSI/IEEE Std 754-1985". IEEE, New York (N.Y.), 1985.

23. "An American national standard. IEEE standard for radix-independent floating-point arithmetic : ANSI/IEEE Std 854-1987". IEEE, New York (N.Y.), 1987.

24. Yakovlev A.G. On the classification style of programming. *In "All-Union Conf. on Actual Problems of New Information Technology Development*

and Inculcation, Tallinn, March 29-31, 1989, Part 1", pp. 69-71. Moscow, 1989 (in Russian).

25. Yakovlev A.G. The classification approach and programming technology. *In "Programming Technology of 90-th : International Conference-Fair, Kiev, May 14-17, 1991"*, pp. 61-63. Kiev, 1991 (in Russian).

Moscow Institute for New Technologies
in Education
Nizhnyaya Radishchevskaya 10
Moscow, 109004
Russia
E-mail: yakovlev@ globlab.msk.su

109004, Москва
ул. Нижняя Радищевская, д. 10
Московский Институт новых
технологий образования
Факс: (095)315-08-08
Эл. почта: yakovlev@globlab.msk.su