Interval Computations
No 4, 1993

Multiaspectness and Localization

Alexander G. Yakovlev

One of the main objectives of interval mathematics is to produce an interval that contains the solution to a given mathematical problem (i.e., the Cauchy problem). There exist methods that produce ellipsoids, parallelepipeds, and other types of sets that contain the same solution. To get a better estimate, it is natural to consider the *intersection* of sets belonging to different classes of mathematical objects as a localization of the desired solution. In this paper, we propose new data types that help to implement this idea in programming. The use of these data types also helps to organize an interface between sumbolic and numerical computations, and to parallelize the resulting program.

Многоаспектность и локализация

А. Г. Яковлев

Одной из основных целей интервальной математики является нахождение интервала, содержащего решение некоторой математической задачи (например, задачи Коши). Существуют методы, использующие эллипсоиды, параллелепипеды и другие типы множеств для локализации одного и того же решения. Для получения лучшей оценки такого решения естественно рассматривать *пересечение* множеств, относящихся к различным классам математических объектов. В этой связи предложены новые типы данных, предназначенные для программной реализации данной идеи. Использование подобных типов данных помогает также организовать интерфейс между символьными и числовыми вычислениями и создает возможность распараллеливания порожденной программы.

1 Introduction

First, we will explain (informally) what we mean by a *localization task*. By this, we mean the task of describing a set that contains a solution to a given mathematical problem. The smaller this set (i.e., the closer its boundaries to the solution), the better is our knowledge. A procedure that computes such a set will be called a *localization*, and the computations that implement this procedure will be called *localization*, or *localizing* computations.

In the following text, we will use the concept of a *locus* introduced and used in [1, 2]. A locus is a pair (x, X), where X is a (known) set called a *shell*, and x is an (unknown) element of X called a *kernel*. In these terms, a *localization* procedure, or a localizing computation, means *finding of a shell* for a kernel that is defined as a solution to a given mathematical problem. Shells are usually chosen from a fixed class of sets. To process data given as shells, we need to define some operations with the shells (and maybe some relations between the shells). A class of sets (= potential shells), with operations and relations defined for these sets, will be called an *aspect*. Interval computations, being understood as computations in the interval arithmetic, can be viewed as a particular case of localizing computations: here, shells are intervals, and operations are standard operations of interval arithmetic.

One and the same element x can belong to shells of different type. To describe such cases, we will define a *multiaspect locus* as a tuple $(x, X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are different shells of x belonging to different aspects $A_1, \ldots, A_n$. Using multiaspect loci enables the user to move from working with *simple shells* (for example, ellipsoids and polygons on a two-dimensional plane, cones and parallelepipeds in a n-dimensional space, etc.) to using complex objects formed as *intersections of simple shells*. So, it is desirable to compute a multiaspect locus $(x, X_1, \ldots, X_n)$ that contains x, and then conclude that x belongs to the intersection $X_1 \cap \cdots \cap X_n$. In other words, we must develop localizing computations for multiaspect loci.

In principle, localizing operations can be performed aspectwise (independently for each aspect). However, it is often better to use the results of computing a shell of one type to improve the shell of another type. For example, let a multiaspect locus (x, X_1, X_2) be given, where x is a complex number, X_1 and X_2 are respectively a rectangle and a circle on the complex plane (the arithmetics of rectangles and circles are described in [3]). The

intersection of X_1 and X_2 contains x. Therefore, as a rectangular shell for x, we can take not only X_1, but also an arbitrary rectangle that contains $X_1 \cap X_2$. In particular, among all such rectangle, we can take a rectangle X_1' with the smallest possible area. In general, X_1' is smaller than X_1, and this leads to a better estimate for x. Similarly, instead of the original circular area X_2, we can take any circle that contains the intersection $X_1 \cap X_2$. In particular, among such circles, we can choose the one X_2' with the smallest possible radius. In general, this will be a better estimate for x that the original circle X_2.

We want to have an easy way to program computations with multiaspect loci, and we want the resulting programs to be efficient. For these two goals, we need an appropriate language support. In the following text, we present special language constructions that enable the user to write short, efficient, and easy-to-write programs for situations with various computational resources. As a basis for these constructions, we take the general purpose language Ada.

2 Multiaspect data types

In the desired language for programming localizing computations, multiaspect loci must be represented by corresponding multiaspect data types. We call a data type T *multiaspect* if a finite list of data types (called *basic*) is fixed, and every object of type T has representations belonging to one or more basic data types. These basic types will be called *aspects* of the multiaspect type.

The relation between multiaspect types and multiaspect loci is evident: *basic types* (that are aspects of a multiaspect type) serve as representations for *classes of shells* (which are aspects of multiaspect loci). Generally, it is convenient to use multiaspect types in the situations when a frequent joint use of corresponding operations and relations over different types of elements is expected. In this respect, multiaspect types are a very convenient means for representing multiaspect loci. For example, let us consider two aspects: rectangles on the complex plane, and circles on the complex plane. On both aspects, similar operations are defined: namely, basic arithmetic and theoretic-set operations.

We can define different kinds of multi-aspect types:

- If an object of a multiaspect type can have only one representation (belonging to one of the basic types), then this multiaspect type will be called *disjunctive* (because an object of this type either has a representation of type A, OR a representation of type B, OR etc.).
- If an object of a multiaspect type can have several different representations from the different basic types (i.e., it belongs to the type A, AND to the type B, AND etc), then such a type will be called *conjunctive*.

There are two known ways to represent a *disjunctive* type:

- *static*, in which the basic type is chosen during the compilation (based on the context) and never changed later;
- *dynamic*, in which the basic type is chosen during the computation time.

The so-called united mode of the **Algol 68** language presents an example of a disjunctive multiaspect type. In most other languages such types are modelled in an indirect way: by means of pointers, variant records, and so on. In the **Ada**-like notation, a disjunctive type can be described by a construct of the following form:

```
type TYPE_D is TYPE1 or TYPE2 or TYPE3; -- the type is
                                           composed of three
                                           basic types.
```

The computer representation of a *conjunctive* type is usually static. Namely, conjunctive types can be easily modelled by records with fields belonging to corresponding basic types. A "legitimate" **Ada**-like representation would be of the following form:

```
type TYPE_C is TYPE1 and TYPE2 and TYPE3; -- the type is
                                             composed of the
                                             same three basic
                                             types as TYPE_D.
```

In the high-level language that we are describing, we do not need access to individual components of the multi-aspect type. All operations that are defined on an object will be applied (aspect-wise) to *all* the components. The fact that some characteristics (components, operations, etc) of the complex data type are not directly accessible is called *encapsulation.* So, in programming terms, in our representation, we will be using encapsulation.

We want to be able to apply the same operation (e.g., +) to all components. Therefore, it is highly desirable that the same symbol be used for operations applied to different components. The possibility to use the same symbol (e.g., +) to describe addition of numbers, addition of vectors, addition of intervals, etc, is called *overloading.* This possibility exists in Ada, and this is one of the reasons why we chose Ada as an example.

3 Using multiaspectness for organizing a symbolic-numerical interface

Multiaspectness may play an important role in the organization of a symbolic-numerical interface. The point is that multiaspect types can include as basic not only numeric, but also analytical types. For example, let the conjunctive type LOCUS1 be described as

```
type LOCUS1 is EXPRESSION and INTERVAL
```

where an element of the basic type EXPRESSION is either a string or a list of characters (such data types have been proposed, e.g., in [4, 5]), and an element of the INTERVAL type is a pair of floating point numbers. Such a multiaspect type may be used for simultaneous processing of both an analytical expression and of an interval that represents a numerical value of this expression. The simplest way to define operations with data of this type is to define them componentwise. But to make interval estimates better, one can define and implement operations over elements of the LOCUS1 type in such a way that after each operation, the resulting analytical expression will be used to compute a new interval estimate for the desired variable, and as a resulting interval component, we will take the intersection of the interval obtained by applying interval operations, and the interval, that was

computed using the analytical expression. To do this, one must define an element of the EXPRESSION type as a pair consisting of the expression itself, and of the bounds for all the variables from this expression.

For example, suppose that after some arithmetic operation has been applied to both EXPRESSION and INTERVAL parts of the object y, we have $x * x + 1$ as an EXPRESSION part (with $(-\infty, +\infty)$ as the range of possible values of x), and $[-0.5, 2]$ as an INTERVAL part. Assume also that we have no knowledge about the exact value of x; in other words, that the range of possible values of x coincides with the entire real line $(-\infty, +\infty)$. For x from this range, the range of possible values of $x * x + 1$ is $[1, +\infty)$. So, to get the refined interval estimate for y, we can take into consideration that y belongs to $[-0.5, 2]$, and that y belongs to $[1, +\infty)$. Therefore, y belongs to the intersection $[1, 2] = [1, \infty) \cap [-0.5, 2]$. This intersection is then chosen as the new (refined) value of the INTERVAL part of y.

The main idea of this improvement is that the EXPRESSION type contains direct information about the kernel, while the INTERVAL type contains information about the shell localizing this kernel. Evidently, direct information about the kernel can be used to refine its shell.

4 Conversions of multiaspect types

In the previous text, we considered the case when all operands are of the same type (e.g., all rectangles, or all described as a pair of a circle and a rectangle, etc). However, in some cases, we need to compute an arithmetic expression for which different operands are described by different multiaspect types.

In the following text, we will show how this can be done. We will not present the complete solution to the problem, but we will explain the main ideas that (we hope) will help. All explanations will be carried on the example of arithmetic operations.

In our considerations, we will assume that for every pair of aspects (basic data types) we already know how to transform estimates of one type into estimates of another type (e.g., in the complex example, we know how, given a rectangle X_1, to construct a circle X_2 that contains X_2; and we also know, how to transform a circle X_2 into a rectangle X_1 that contains this circle).

4.1 Conversions in operations with disjunctive types

Let us first consider the case when all the variables are of disjunctive type. The problem arises only if the variables involved in some operation have different basic types. Clearly, such an operation requires transforming variables into a common basic type. This may be performed in different ways. For example, let's consider the following program fragment:

```
A, B, C : TYPE_D; -- see above the type definition
      D : TYPE1;
begin
      ........
      D := A * B + C;
end;
```

Assume that during the run time (and before computing D) the multiaspect variables A, B and C have obtained values of different basic types, so before we compute D, we must perform some data conversions. There are two possible conversion tactics:

1)
- One of the variables A and B is converted to the type of the other.
- Multiplication is performed.
- The result of multiplication is either converted to the type of C, or C is converted to the type of the result of the multiplication.
- Addition is performed.
- The result of addition is converted to the type of D, and the assignment is performed.

2)
- A, B, and C are converted to the type of D.
- The expression is computed, and assigned to D.

Even in this example, we can use several different computation tactics, and it is difficult to guess which method will lead to a better estimate. For complicated expressions, the number of possibilities grows exponentially.

Therefore, it may be quite difficult (and sometimes even impossible) to determine a priori a tactic leading to the most precise result. The tactics allowing to avoid exceptions like divide-by-zero, or overflow (or at least to minimize the number of these exceptions) are also extremely difficult to choose.

We also face the following dilemma: should we choose a conversion to the type whose operations are quickly executable but not quite sharp, or a conversion to the type with less sharp but faster operations?

Thus, an essential feature of disjunctive types is the impossibility of choosing a priori the best tactic of computing expressions.

There is one important exception to this general rule: the conversion tactics is easier to choose if one of the operands is of the analytical expression type, and the objective of our computations is to get the most precise result (so that run time is not our main concern). In this case, it is necessary to use an analytical representation type as much as possible. The reason for such a tactic is that, e.g., in interval computations, overestimation is caused by the dependency of the operands; the trivial example is $x * x - x * x$: it should be 0, but for, say, $x \in [0, 1]$, interval arithmetic leads to $[-1, 1]$. If we use analytical expressions, then we will get the desired 0 as the result.

When one of the operands is an analytical expression, and another one is an interval, then we would prefer to get the result in a semi-analytical form, and postpone the conversion to an interval for as long as possible. There are several ways to do that, among them:

- analytical transformations based on the application of the subdistributivity property and other properties of the interval arithmetic and its generalizations (such methods are described in [6]);
- a posteriori interval analysis [7];
- Hansen's arithmetic [8], etc.

Let us consider an example. Assume that we have defined a *disjunctive* type LOCUS2 composed of the same basic types as LOCUS1. Assume also that for every expression of the EXPRESSION type, for each variable from this expression, we are explicitly given the range of its possible values. In the following program, a constant in quotes (e.g., '$[1, 2]$') belongs to the

type EXPRESSION, and a constant without quotes belongs to the type INTERVAL. We consider the following program:

```
type LOCUS2 is EXPRESSION or INTERVAL;
A, B, C, D, E : LOCUS2;
    F : INTERVAL;
begin
    A := '[1, 2]';   -- A = '[1, 2]'
    B := A + [2, 3]; -- B = A('[1, 2]') + [2, 3]
    C := B - [3, 4]; -- C = A('[1, 2]') + [2, 3] - [3, 4]
    D := A * [4, 5]; -- D = A('[1, 2]') * [4, 5]
    E := D / [5, 6]; -- E = A('[1, 2]') * [4, 5] / [5, 6]
    F := C - E;      -- F = A('[1, 2]') + [2, 3] - [3, 4] -
                          - A('[1, 2]') * [4, 5] / [5, 6] =
                          = A('[1, 2]') * [0, 1/3] + [-2, 0] =
                          = [-2, 2/3]
end;
```

In a comment part of this program, we present the results of executing each of its operators (we assume that if different types meet, then the analytical expression type "prevails"). Computing F begins with moving A outside the parentheses (to achieve that, we use the subdistributivity property). We move A outside to simplify the expression in the right-hand side of the assignment statement. Then, the constant '$[1, 2]$' is converted to the INTERVAL type, and the expression is computed according to usual rules of interval arithmetic.

Note that if instead of converting everything to an EXPRESSION type, we choose the conversion to the INTERVAL type, then the values of B, C, D, and E would be correspondingly equal to $[3, 5]$, $[-1, 2]$, $[4, 10]$, and $[2/3, 2]$, and F would be equal to $[-3, 11/3]$, which is a less precise result than $[-2, 2/3]$ obtained by the first method. Thus, this example shows that retaining the analytical form as long as possible is reasonable.

4.2 Conversions in operations with conjunctive types

Let's now consider the case when some of the operands belong to a conjunctive multiaspect type.

It is sufficient to consider the case when *all* types are conjunctive, because we can consider a disjunctive type as a one-aspect conjunctive one.

If both operands consists of components that belong to the same list of aspects, then we simply perform componentwise operations with corresponding components. If two operands are represented by shells from different lists of aspects, then each operands must be "completed" to an element composed of the *union* of basic types of operands (so that each element will be represented by shells of all necessary types). The completion is performed by means of conversion of representation of given basic type into the new basic types. If, later on, we must assign the result to a multiaspect variable with a fewer number of aspects, then the assignment is performed only for corresponding aspects, and all the other components are simply deleted. As an example, consider the following program:

```
type LOCUS3 is EXPRESSION and RECTANGLE;
type LOCUS4 is EXPRESSION and CIRCLE;
type LOCUS5 is RECTANGLE and CIRCLE;
    A : LOCUS3; B : LOCUS4; C : LOCUS5;
begin
    C := A + B;
end
```

where the EXPRESSION type is the same as above (with the only difference that the variables involved in the formulas can now denote complex intervals as well), and the RECTANGLE and CIRCLE types present respectively rectangular and circular complex intervals. The above program can be executed as follows:

- First, we describe the type that is the union of the basic types of operands:
  {EXPRESSION, RECTANGLE} $\cup$ {EXPRESSION, CIRCLE} = {EXPRESSION, RECTANGLE, CIRCLE}.
- Both A and B are "completed" to this type:
  - the rectangle A is inscribed into a circle, and

  - the circle B is inscribed into a rectangle (with sides parallel to the coordinate axes).

- Next, addition is performed. As a result, we get a value of the union type.

- Finally, we transform the result of addition into a C type (by deleting its analytical expression component), and assign the resulting value to the variable C.

If an expression with conjunctive type operands involves more than one operation, then a similar sequence of steps is performed for every operation.

In some cases, a missing component (that needs to be computed for completion) must be computed based on components belonging to *several* basic types.

5 Comparison of convenience and efficiency of different multiaspect types

In the framework of localizing computations, conjunctive types turn out to be more convenient for a user than disjunctive ones. Simultaneous accessibility of several representations enables the user (at any stage of the computation process) to make a mutual refining of shells corresponding to different components of the same element. This refining can be done both by taking their intersection, and by comparing their analytical representation with numerical ones (see above).

Another advantage of conjunctive types (important from the implementation viewpoint) is the possibility of *parallel processing* of components corresponding to different basic types. This parallelization enables a user to reduce the time that is necessary to compute a result with a given accuracy by using additional available computational resources. A program for processing a n-aspect conjunctive type is thus regarded as subdivided into n processes. For example, a program fragment

```
A, B, C : TYPE_C; -- see above the type definition
begin
```

```
    C := A + B;
end
```

can be regarded as divided into three processes of the form

```
A, B, C : TYPEn; -- where n = 1, 2, 3
begin
    C := A + B;
end.
```

If we want to refine C, we must allow these processes to interact. The more frequently they interact, the more precise results each of them computes. However, the interaction itself requires time and slows down each of interacting processes. Depending on the requirements on speed and accuracy of the computations, we can use various tactics of organizing such an interaction. Various tactics of parallelization are discussed in [2, 9] (in connection with the so-called *wave* computations).

Summarizing: The use of conjunctive types is more convenient for the user, and leads to a higher accuracy of the final result. For time efficiency, the results depend on whether we can parallelize:

- For *sequential* (non-parallel) computations, conjunctive types are more time-consuming than disjunctive ones.
- If we have an (appropriately organized) *parallelization*, then conjunctive types are as (or almost as) efficient as disjunctive ones.

6 Other applications of localizing computations

It should be pointed out that multiaspect types can be used not only for representing shells, but for other purposes as well. For example, triplex intervals [10] can be described as follows:

```
type TRIPLEX is INTERVAL and MAIN
```

where the element of the basic type INTERVAL is an interval represented by a pair of floating point numbers, and MAIN is a number from this interval.

The INTERVAL component is interpreted as an interval that contains all possible values of the unknown physical quantity x, and the MAIN component is a value that x *can* take. For example, suppose that we have measured a physical quantity x, and the result is 0.5. We know that the measuring device has a guaranteed accuracy ± 0.1, and we know that the result 0.5 is possibly correct. Then, the result of this measurement can be represented by a triplex $([0.4, 0.6], 0.5)$. The idea behind adding 0.5 is that $[0.4, 0.6]$ can be an overestimation; the actual interval (corresponding to the accuracy of this measuring device) can be smaller that $[0.4, 0.6]$. In view of that, in addition to the interval, it would be nice to know where inside this interval the actual interval of uncertainty may be located. The fact that 0.5 is a possible value means that the actual interval cannot be in $[0.4, 0.49]$, or in $[0.51, 0.6]$: it has to include 0.5.

In general, multiaspect types are convenient to use in all cases where frequent joint use of corresponding operations and relations over elements of different types is expected.

7 Conclusion

The present paper has described only a few applications and advantages of multiaspectness. The idea of the multiaspect approach like that of the object-oriented approach is certainly substantially richer than it may seem after considering one or two particular implementations of this idea. Multiaspectness is worth developing. The author will be glad to learn about the work of other researchers in this direction.

References

[1] Yakovlev, A. G. *Loci and localizing computations.* In: "Conf. on Interval Mathematics, May 23-25, 1989", Saratov, 1989, pp. 54–56 (in Russian).

[2] Yakovlev, A. G. *On some possibilities in organization of localizing (interval) computations on electronic computers.* Inf.-operat. material (interval analysis), preprint 16, Computer Center, Siberian Branch of the

USSR Academy of Sciences, Krasnoyarsk (1990), pp. 33–38 (in Russian).

[3] Alefeld, G. and Herzberger, J. *Introduction to interval computations.* Academic Press, New York etc., 1983. — Russian translation: Mir, Moscow, 1987.

[4] Stetter, H. J. *Inclusion algorithms with functions as data.* In: Kulisch, U. and Stetter, H. J. (eds) "Scientific computation with automatic result verification: Papers presented at a conf., Sept. 30 — Oct. 2, 1987, Karlsruhe", Springer Verlag, Wien — New York, 1988 (Computing, Suppl. **6**), pp. 213–224.

[5] Caplat, G. *Symbolic preprocessing in interval function computing.* Lect. Notes in Comp. Sci. **72** (1979), pp. 369–382.

[6] Matiyasevich, Yu. V. *Real numbers and computers.* Kibernetika i vychisl. tekhnika **2** (1989), pp. 104–133 (in Russian).

[7] Hansen, E. *A generalized interval arithmetic.* In: Nickel, K. (ed.) "Interval mathematics", Springer Verlag, New York etc., 1975 (Lecture notes in computer science **29**), pp. 7–18.

[8] Yakovlev, A. G. *Specific parallelism of localizing computations.* In: "Proc. All-Union Conf. on Actual Problems of Applied Mathematics, Saratov, May 20–22, 1991", Saratov, 1991, pp. 151–158 (in Russian).

[9] Nickel, K. *Triplex-algol and applications.* In: Hansen, E. (ed.) "Topics in interval analysis", Clarendon Press, Oxford, 1969, pp. 25–34.

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