Interval Computations
No 1(3), 1992

WAVE COMPUTATIONS
A TECHNIQUE FOR OPTIMAL QUASI–CONCURRENT SELF-VALIDATION

Eldar A. Musaev

A method with a variable data representation and automatic choice of minimal necessary data representation is proposed. Due to a hierarchical organization of representations, it is possible to avoid concurrent programming and to describe the method with coroutines.

ВОЛНОВЫЕ ВЫЧИСЛЕНИЯ
СПОСОБ КВАЗИПАРАЛЛЕЛЬНОЙ САМОКОРРЕКЦИИ ВЫЧИСЛЕНИЙ С МИНИМИЗАЦИЕЙ ПРЕДСТАВЛЕНИЯ

Э. А. Мусаев

Предлагается метод вычисления с переменным представлением данных и автоматическим выбором минимально необходимого по временным затратам представления. Благодаря иерархической организации представлений удалось избежать параллелизма и изложить данный метод в рамках сопрограмм.

One of the most serious problems of interval analysis is the possibility that continuation of computations is impossible because two values to be compared happen to be incomparable. The simplest example is the computation of roots of a quadratic equation:

```
1 Seg $a, b, c, d$;   $Read\ ((a, b, c));\ d := b * b - 4 * a * c$;
2 If $d < 0$          Then $Print(\text{“}No\ roots.\text{”})$
3 Elif $d = 0$        Then $Print((\text{“}One\ root:\text{”}, b/(2 * a)\ ))$
4 Else                     $Print((\text{“}Two\ roots:\text{”}, (b + Sqrt(d))/(2 * a),$
5                                              $(b - Sqrt(d))/(2 * a)))$
6 Fi
```

It is evident that, if **Seg** is an interval data type and $d = [-1, 1]$, then choice of the branch in line 2 is already impossible, and even more clearly so in the line 3, where comparison would not make sense in practice even if d were an ordinary floating-point value. In this case, indeterminacy in the original data prevents us from taking any action. However, when a, b, and c have been obtained as a result of computations, repeating the computations with higher precision usually seems to be an attractive idea. On the other hand, it is impossible to determine a priori what precision is necessary to get the proper result, so this leads to loss of time when numerous repetitions or too high a precision is used. The idea that a computational process can choose the necessary precision is suggested in [1]. Let us use the term *program shell* to refer to this program's computational process (in sense of **UNIX** or **Modula-2**), together with an associated data representation. Suppose that the program consists of the several shells which differ only in the representations used. In the implementation suggested here, the shell with the simplest representation is initialized first. If incomparable values occur, then this shell is frozen, and control is transferred to a shell with a more complex representation (in the case of concurrent processes it could be initialized immediately, but have lower priority). When this shell comes to the point with incomparable values, it corrects them, then activates the previous shell, but if the values are still incomparable, a third shell is initialized to compute the values more accurately. In this scheme, the low level shell makes a path for the complex and expensive high level shells, computing, when possible, directions for IF and CASE operators, as well as iteration counters for loops. On the other hand, the high level shells correct the values computed by the low level shells. This is why we use the term “wave computations”: the points of control in the shells run one after another like waves on a sea.

An implementation of this idea is considered here. It is described using a language similar to **Algol-68**, with elements of **Pascal** and **C**. It could be also described in **C++** or other object-oriented languages.

When concurrent processing is actually available, it is not necessary to freeze all high level processes. It is sufficient to have a guarantee that a process with a higher level will never overtake any lower-level process. (i.e. it is necessary to process rendezvous points properly.) A low-level process should have higher priority than a higher-level process, so a level-0 process is frozen when and only when it requires the help of the level-1 process, a level-1 process frozen when and only when it requires the help of the level-2 process, or it is at the same point as the level-0 process, or the level-0 process is presently running and there are no processor resources.

1. Language conventions

The language to be used to describe the method is similar to **Algol-68**, with the following exceptions:

1. The language contains the enumerable data types of **Pascal.** Two of them will be important here. The first is

   Mode Trin = (*True*, *False*, *Unknown*);

   The Trinary data type is used together with the ordinary Boolean type. The common *True* and *False* are accompanied by the new value *Unknown*, used in the situations like $[-1,1] > [0,0]$. For more details about this type, see [2].

   Mode Level = (*Single*, *Double*, *Multi*);

   Level *FirstLevel* = *Single*, *LastLevel* = *Multi*;

   This data type is used to specify the representations and shells. It should contain the names of all representations in order from the simplest to the most complex one. In this example, three representations are assumed: a single precision floating-point based representation, a double floating-point based representation, and a multiple precision representation. More precisely, this type can be replaced by the following set of descriptions:

```
Mode Level = ShortInt;
        Single = 1,
        Double = 2,
        Multi  = 3;
 Level First Level = Single, LastLevel = Multi;
```

2. The language contains a numerical data type which is a base type, and which can be described by:

   Mode Num = **Struct** (**SingleSeg** s,
   DoubleSeg d, **MultiSeg** m);

   All variables of this type are static and common to all shells. All other variables, except global variables, are dynamic (automatic), and every shell has a unique copy of them.
3. A library to support coroutines is present and consists of:
   Activate (**Corout**) – This procedure activates the specified coroutine, and freezes the current coroutine.
   Initialize (**Ref Corout, Proc Void**) – This procedure links the body and the name of coroutine, and initializes it.
   Ready () - This procedure creates a signal that initialization is complete and transfers control to the parent process.

These procedures organize the work of the coroutines as follows. The parent process links the names of coroutines (variables of type **Ref Corout**) with the bodies (procedures of type **Proc Void**) using the *Initialize* procedure. In particular, *Initialize* transfers control to the body of each coroutine, while *Ready* returns control upon completion of the initialization. After that, the parent process initiates one of the coroutines using *Activate*. The initiated coroutine then becomes active. During execution, the active coroutine can transfer control to another coroutine using *Activate*; in this case it is frozen, and the coroutine pointed to by *Activate* becomes active.

2. Implementation with coroutines

It is natural to implement the wave computations support as a separate module which contains all necessary procedures and data. First, we need a pointer to the active shell and an array of coroutines:

```
Level GlobLev := FirstLevel;# Global Level#
[Level ]Corout Shell;        # Shells      #
```

Additionally, we need a multilevel queue, defined as follows. We require a queue with a special pointer for every level. An item leaves this queue when the last pointer attains and surpasses it. An item is placed in the queue when the first pointer is used to determine its value. There are only two operations associated with this queue. The first is to examine the value of the current item at the current level, while the second is to change the current value and move to the next item. (If the next item does not exist, then create it by the some rules.) Data for the multilevel queue takes the form:

```
Struct([Level] Int p                # Pointers    #,
       [10000]Struct( Trin r,  Level l)q)
       GlQ                          # Global Queue #;
ptrs of GlQ := (0,0,0); r of q of GlQ := Unknown;
```

The following procedures implement the actions listed above:

```
Proc SeeQueue = Trin :
        r of q[ p[GlobLev] of GlQ]of GlQ;
Proc SetQueue = ( Trin t, Level l)Void :
        If q[p[GlobLev] of GlQ] of GlQ := (t, l);
           p[GlobLev] of GlQ = 10000
        Then p[GlobLev] of GlQ := 1
        Else p[GlobLev] of GlQ+ := 1
        Fi
```

The procedures JumpUp and JumpDown are also useful; these activate a shell one level higher or lower if present, respectively. Now, if the program of computations is written as procedure *Run*, the parent process body should be the following:

```
( For i From FirstLevel To LastLevel
   Do GlobLev := i; Initialize(Shell[i], Run)
   Od; GlobLev := FirstLevel; Activate(Shell[GlobLev]) )
```

The body of *Run* should be started from the call of the *Ready* procedure. As it was mentioned above, the numerical data are common to all shells, while other data are individual. Arithmetic operations are executed only for the item of a structure corresponding to the current level.

e.g.:

```
Op + = ( Num a,b) Num : ( Num r;
      ( GlobLev ! s of r := s of a + s of b,
                  d of r := d of a + d of b,
                  m of r := m of a + m of b); r)
```

The comparison operations use parameters in a way similar to **Algol-60** parameters, i.e. as quantities which are computed every time the parameter is used and which are never computed if the parameter is not used. Thus, values computed at the lower levels are taken from the queue, while expressions used as arguments in comparisons are not calculated. The comparison operations are used as points of rendezvous, and appear to be critical points of a program. It is also possible to combine these with the operation of converting a **Num** data type to integer to make a special function for the correction of a numerical value. This is easily implemented by a small modification of the queue, in which a queue item is not only trinary, but may also be integer or numerical (interval). All comparison operations are similar, so we'll consider only the 'Greater' operation:

```
Op > =      (Num a,b) Trin:
    If Trin f = SeeQueue;
            f <> Unknown             # Already computed ? #
    Then    SetQueue(f,GlobLev); f
    Elif    # This level has not been computed yet #
            # Correct all values for a level down ! #
            RecomputeAll;
            # Compute at this level              #
            Trin f = ( GlobLev ! s of a > s of b,
                  d of a > d of b, m of a > m of b );
            f = Unknown            # Not success ? #
    Then    # This level is insufficient #
            # A level up and compute ... #
            JumpUp; JumpDown; SeeQueue
    Else    # Success, now save ... #
            SetQueue(f,GlobLev); JumpDown; f
    Fi
```

The only procedure not given in detail here is RecomputeAll, i.e. the correction of all values for one level down. This routine transfers the values computed at the higher levels to the lower levels. It could be implemented as an intersection of every value at the present level with the corresponding value at the previous level. Since the comparison operations are the points of rendezvous the values have the same meaning. Of course, the results should be stored at both levels, e.g. if the values are $[-1, 3]$ and $[2, 5]$, the result would be $[2, 3]$, which is better than both source values. The full list of variables can be obtained using the technique described in [2] or using object-oriented programming.

Note that coroutines are not even necessary. Let us suppose that every variable (not only numerical, but also non-numerical ones) is an array with index of type **Level**. At each point, only the items with index *GlobLev* are used. Also, exclude from the language automatic variables or **Algol**-like block structure (when every block is used for declaration and allocation of the automatic variables). Finally, two procedures, similar to the procedures presented in some implementations of **C**, are needed. The first one is *SetJump*, which sets the dynamic label to make a subsequent jump to this label, and the second one is *Jump*, which carries out the jump to the dynamic label created by *SetJump*. The *JumpUp* and *JumpDown* procedures can be written as:

```
Proc   Label SetJump;
Proc   (Label ) Void Jump;
Label  Returns [ Level];
Macro Proc JumpUp = Void :
 (     GlobLev = LastLevel ! print("Indefinity error.") !
       Returns[GlobLev] := SetJump;
       GlobLev + := 1; Jump[Returns[GlobLevel]] )
Macro Proc JumpDown = Void :
 (     GlobLev = FirstLevel ! Skip !
       Returns[GlobLev] := SetJump;
       GlobLev − := 1; Jump[Returns[GlobLevel]] )
```

An interesting question for the future research is the possibility of structuring such computations without a hierarchical organization of numerical data types. That could be useful if there are several representations which are approximately similar in complexity, e.g. centre and left-right bound representations of interval numbers.

References

1. Yakovlev A.G., *On some possibilities in the organization of localizational (interval) computations on electronic computers*, Inf.-operat. material (Interval analysis), preprint 16, Computer Centre, Siberian Branch of the USSR Academy of Sciences, Krasnoyarsk, 1990, p. 33–38. (In Russian)
2. Musaev E.A., *The support of interval computations in high-level languages*, Proc. 1-st Sov.-Bulg. Seminar on Numerical Processing Oct. 19-24, 1987, Program Systems Institute of the USSR Academy of Sciences, Pereslavl-Zalessky, 1989, pp. 110--121, deposited in VINITI 21.04.89, 2634-B89. (In Russian)
3. Musaev E.A., *Wave computations in interval analysis*, Proc. Seminar on Interval Mathematics, May 29–31, 1990, Saratov, 1990, pp. 95–100. (In Russian)

St.Petersburg Division of
Steklov's Mathematical Institute
Russian Academy of Sciences
Fontanka 27
St.Petersburg 191011 Russia
e-mail: eldar@lomi.spb.su