# Mixed Computation in Novosibirsk

Mikhail Bulyonkov



# Mixed Computation in Novosibirsk

Mikhail Bulyonkov

Ershov Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences
630090, Novosibirsk, pr. Acad. Lavrentieva, 6
mike@iis.nsk.su

**Abstract.** In these notes, I would like to give an account of the history of mixed computations at Novosibirsk Computing Center and later at A.P. Ershov Institute of Informatics Systems. It is quite possible (and even most probable) that I will not be able to mention all persons and events relevant to the works in this field, but in no way, it is to diminish their significance.

**Keywords:** Mixed computation, partial evaluation, program specialization

## 1. Futamura Projections — Andrei Ershov

Andrei Petrovich Ershov came to the idea of mixed computation from his work in compiler construction, program optimization, and transformation. In his first works on the subject, mixed computation processor, or *partial evaluator* was defined as program processor that has as its input some program representation and part of its input data, and produces as its output transformed program, partial result, and the data demanding further processing.

Next thing Ershov noticed was the fact that compiler's behavior is very similar to that of mixed computation: operations of the compiled program analysis alternate with object code generation. These considerations culminated in discovery of relationships between program interpretation and its translation into object code. For this purpose, they interpreted a partial evaluator in a more restricted sense – without intermediate data and partial result:

$$mix(p, x) = p_x$$

such that

$$p(x, y) = p_x(y)$$

Let us call the last equality a *mixed computation equation*. Let *int* be interpreter of some language *L*, written in the language whose programs can be processed by *mix*. Then

$$int(p, d) = p(d)$$

for each program *p* in *L* and its data *d*. Then, substituting *int* in the mixed computation equation, we get

$$int(p, d) = int_p(d)$$

whence

$$int_p(d) = p(d)$$

Thus, programs $int_p$ and *p* are equivalent, but note that we write them in different languages: *p* in *L*, and $int_p$ in the interpreter language. In other words, mixed computation applied to interpreter implements compilation, and $int_p$ is an object code for *p*.

We can obtain very interesting relations under the assumption that partial evaluator possesses the property of *self-applicability*. For this to occur, we must write it in the same language as the one for which it is intended. We leave to the reader the proof of the following relations:

$$mix_{int}(p) = int_p$$

and

$$mix_{mix}(int) = mix_{int}$$

Thus, $mix_{int}$ transforms a program into object code and hence implements the function of compiler, while $mix_{mix}$ transforms the language semantics, defined in the form of interpreter, into compiler and therefore appears to be compilers generator.

Anybody who had a faintest notion of compilers and interpreters and who became familiar with these relations, at first could not believe in them and looked for some hanky-panky trick. They then became delighted with the beauty, clarity, and depth of their meaning[1]. Today I can clearly imagine Ershov's disappointment at

[1] I heard that Andrei Petrovich made a long-distant call to his son Vasily and for several hours described his new findings. Similar story is told about Valentin Turchin, who also found these relations independently.

having discovered a paper by Japanese researcher Yoshihiko Futamura, who had published these relations back in 1971. Andrei Petrovich called the independently found relations Futamura projections, although the last of the three was not present in Futamura's paper; it serves as an excellent example of professional ethics.

The main approaches to implementing mixed computations appeared as early as the first works on the subject. They include:

1. *Partial evaluation* refers to the immediate execution of all instructions from the source program that depend only on available data. We reduced the remaining instructions and placed them into a residual program.

2. *Generating extension* splits the process of residual program construction into two stages. First, we classify all actions of a program as "available" or "delayed". Then we generate a program that, given as input the available data of the source program, becomes the residual program. People recognized the importance of the static classification of a program's actions as available and delayed much later; they called it *binding-time analysis*, or BTA.

3. *Transformational approach* refers to obtaining the residual program from the source by application of a sequence of so-called reducing transformations such as constant propagation, reduction of expressions and conditionals with constant conditions, loop unfolding, and the elimination of unused computations.

We recognized rather quickly the main problems of mixed computation – a problem of delayed control and related problem of mixed computation termination. In his first works on mixed computation, Ershov neatly disguised these problems, which he acknowledged in a reply to my pointed questions. It had taken years of research and experiments to arrive to a practical implementation from a sketchy idea. However, back then the main goal was to introduce the computer science community to the idea of mixed computation potential and Ershov dealt with it brilliantly. Dozens of times, at various conferences and seminars, in papers and informal talks he spoke about Futamura projections and mixed computation, forming a new school in system programming. His large comprehensive article with many color illustrations that appeared in the popular-scientific journal "V mire nauki"[2] [Ersh84] serves as spectacular example of this activity. The works in the mixed computation field brought Andrei Ershov the prestigious Krylov Prize of the USSR Academy of Sciences.

## 2. Algebra of Mixed Computation — Vladimir Itkin

A rather different background and approach to the study of mixed computation appears in the works of Vladimir Itkin. He gave an impression of a withdrawn, cloistered scientist who valued strictness of proof and theoretical power of result much more that its practical applicability — a typical example of a "theorist"[3]. He wrote several of his first papers in co-authorship with Ershov, but subsequently their co-operation became less close. The motive was Ershov's words that in their joint papers, the general idea was his, while Itkin worked over the technical details. Even subsequent clarifications — *only* the general idea and *all* technical details — apparently were not enough for Vladimir. Although he worked independently, he obtained results and gave a solid theoretical and conceptual basis to other researchers.

The starting point of Itkin's research was a theory of program schemata, oriented in the first place to imperative operator programs over common memory. The most actively used notion was that of explicator — an operator ensuring the required state of some part of program memory. In the simplest case, an explicator is expressed by assignment of given values to variables.

The role of explicators was twofold: on the one hand, being ordinary operators, they were part of the program, so at any moment, the mixed computation process could terminate and the current program declared the residual one. On the other hand, explicators were like current computation points. Vladimir Itkin analyzed and substantiated different patterns of mixed computation process (e.g. end-to-end, dotted, polyvariant). A substantial part of the process was the manipulation with explicators such as the "application" of program statements and merging. Itkin managed to abstract these operations by introducing the set of axioms they have to satisfy. It has lead to the so-called *algebra of mixed computations* [Itkin88].

Vladimir Itkin's last works were of a philosophical nature. Partial evaluation interested him as a fundamental process of a transition from the general to the special. He died tragically in 1991 when he froze to death on his way to church.

[2] The Russian edition of *Scientific American* journal.

[3] I was always amazed by Itkin's "cut and paste" technique of paper writing. If he had to insert or reposition a couple of sentences in his written text, he did it not by editor's markup or by notes on the underside, but literally cut and pasted a piece of sheet with the text to the right place. The result of his effort amounted to a long paper belt picturesquely snaking through his whole office. Upon completion of works, all that remained to be done was cut it into standard-sized pages and give to the secretary for typing.

## 3. Parser Specialization — Boris Ostrovsky

To make the idea of mixed computation more viable, it was essential to find some practical application for it. Ershov proposed this issue to his post-graduate student Boris Ostrovsky as a subject of his Ph.D. thesis. His task was to transform automatically a universal parser for some class of grammars into specialized grammar-oriented parser [Ostr87]. It was clear that parser specialization, although being of independent value, was only the first step on the way to Futamura projections implementation, since we can consider a universal parser as an interpreter, a grammar as an interpreted program, and the input string as its data. From this point of view, only the first projection was at issue and the task of achieving partial evaluator's self-applicability was out of question.

Ostrovsky based his partial evaluator on the transformation approach. To be more exact, it was not a proper partial evaluator, but rather a transformation machine with an ad hoc set of transformations. The process of transformations application was similar to the process of Markov's normal algorithms execution. Transformation markers labeled the text of a universal parser. Usually the process started with a single marker of initial transformation placed before the first program instruction. A non-deterministic iteration of marker selection and application of its transformation followed to the construct labeled by it. The transformation resulted in a program modification and a placement of new markers. The process repeated until no marker remained.

For each universal parser one had to develop the set of transformations individually. In fact, there was a separate partial evaluator for each class of grammars. However, even this approach was technologically justified. First, the basic transformation set was reusable, which guaranteed the ad hoc parser correctness. Second, they could use a single partial evaluator, after being "tuned" once, for a number of grammars. Third, the system supported sophisticated tools for grammar transformation such as regular part extraction and post-processing tools applied to residual programs.

The mixed computation system developed by Ostrovsky was one of the most - if not the most - advanced for those days. Unfortunately, the routine of teaching at the remote university kept Boris from active continuation of the work in this field.

## 4. The Limanchik Summer School on Mixed Computation

In my opinion, research on mixed computation reached true national level after the conference on mixed computation held at the "Limanchik" summer camp of Rostov State University in 1983. The conference was an out-and-out success! It afforded the opportunity to meet and establish good professional contacts to all active researchers in this field. There is no point in discussing the whole conference program; instead, I will note only two memorable scenes.

Viktor Kasyanov gave a talk on the reducing program transformations. The set of transformations was wide enough and constituted a substantial part of the SOCRAT system aimed at processing FORTRAN programs. One of the system's features was the fact that transformations were based not only on a program's text, but also on user annotations. We considered these annotations as "a priori" true statements about a program that could both reduce complexity of necessary analysis and increase applicability of transformations. During the ensuing discussion, Nikolay Nepeivoda took floor and declared in his usual offhand manner that not a single program exists to which one could apply such transformations. Later, it turned out he meant to say that no programmer would deliberately write a branch instruction with an always-true condition. However, in the case of automatically generated programs, application of such transformations can be very promising.

In another discussion Sviatoslav Lavrov cast doubt on practicality of Futamura projections for production compilers, since they cover only the most simple and well-studied syntax-directed part of the process. The most interesting problems of compilation such as register allocation or common subexpressions elimination have origins in neither a partial evaluator nor an interpreter; therefore, they cannot appear in automatically obtained compiler[4].

## 5. Polyvariant Mixed Computation — Mikhail Bulyonkov

Another Ershov's post-graduate student, Tatiana Shaposhnikova, suggested the idea solving the problem of non-termination of interpreter specialization. When she told me about her idea for the first time, I was unprepared to

[4] N. Jones [Jones88] later formulated and researched a similar but more abstractly stated problem: "Can program specialization yield more than linear acceleration?" Here we suppose that complexity measurement takes into account only the delayed data size, and the available data size is constant.

perceive it, since Tatiana's suggestion led to an introduction of unstructured *goto* statements. In my opinion, it posed a problem, because she essentially based the existing mixed computation patterns on program structuring as in the propagation of available information through delayed conditionals.

I succeeded in finding sufficient conditions of specialization process termination. If the binding-time analysis classifies variables and, therefore, program instructions statically (i.e. if we declare a variable static, it remains so at any moment of specialization), and the range of each available variable is finite, then each branch of polyvariant specialization will inevitably lead to an already passed state. In certain sense, computation turns out to be "shut" inside a matrix with rows corresponding to initial program instructions and columns corresponding to available memory states. Each cell of this matrix contains the corresponding reduced instruction.

Andrei Petrovich grasped this idea with enthusiasm[5] – it has become the key to implementation of self-applicable partial evaluator Mix suitable for carrying out all three Futamura projections. To reduce "technical" problems, we designed a special imperative language IL whose set of basic operations included among others the operation of expression reduction. We presented the methodological comprehension of the obtained results in our joint paper [BulErsh86].

Unfortunately, we were a little late again. Just a few months earlier, Peter Sestoft, under the supervision of Professor Neil Jones from DIKU, showed the successful implementation of all Futamura projections for a small subset of the functional language LISP. In turn, they were surprised to discover belatedly the paper [Bul84] that described the technique they independently found during their partial evaluator implementation. However, these small disappointments became the starting point of a long-term collaboration, mutual visits, and healthy competition between DIKU and Novosibirsk group.

## 6. Partial Evaluation and Compilation Phases — Guntis Barzdin

Guntis Barzdin came to Novosibirsk to negotiate the possibility of his post-graduate study at Latvian University with Ershov as his scientific adviser. After having settled his affairs, Guntis paid me a visit and was very surprised to find me being only a few years older than he was. He had read my papers and pictured me as reputable man of science that was contrary to fact. During our discussions of mixed computation and Futamura projections, Guntis noted that they used only program specialization, while ignoring the possibility of obtaining intermediate data. It would be interesting to look at the projections in more general interpretation of mixed computation.

For the sake of simplicity let mix computation be represented by two processors: specializer *spec* generating residual program $p_x$, and partial evaluator *peval* generating intermediate data $x_p$:

$$spec(p, x) = p_x$$
$$peval(p, x) = x_p$$

such that

$$p(x, y) = p_x(x_p, y)$$

for every $y$. On the one hand, if *peval* is trivial and $x_p$ is always empty, these relations degenerate into ordinary mixed computation equation. On the other hand, if the result of available computation is expressed as intermediate data, we can consider the task of mixed computation to be converting the data $x$ into another – in some sense, more effective – representation of $x_p$.

Substituting in the above equations $p$ with *int*, $x$ with program $p$, and $y$ with the data $d$ of program $p$, we have

$$spec(int, p) = int_p$$
$$peval(int, p) = p_{int}$$

such that

$$int(p, d) = int_p(p_{int}, d)$$

Besides converting a program into some intermediate representation, mixed computation also generates an interpreter of this representation. In the case of degeneration, generation of the interpreter $int_p$ does not depend on program $p$ at all, but only on the binding-time analysis result. The difference between such an interpreter and the initial one is the following: Whenever the initial interpreter performs computations over the initial data, the residual interpreter extracts the results of these computations from data structure $x_p$. The residual interpreter has some auxiliary variables pointing to the current data element of $x_p$. The algorithm we developed was the mirror image of polyvariant specialization, where the specialization process generated the next residual program

[5] It was Ershov who translated into English (or to be honest — rewrote in English) my paper [Bul84] that was later extensively cited.

fragment; the partial evaluator generated intermediate data element by putting there the values of this fragment's available expressions.

If available, we could split data into two sufficiently independent parts. There appears the possibility of a mixed strategy when we express one part of available computations in residual program and the other in intermediate data. The resulting residual program can be in turn specialized against the intermediate data. This strategy when applied to compilation problems corresponds to sequential *compilation phases*.

The main advantage of partial evaluation compared to specialization is that a more compact result occurs with the same volume of available computation because we no longer need to "decorate" computed values with the initial program fragments. On the other hand, we pay for this with the residual interpretation expenses [BarzBul88]. We did not bring the idea of polyvariant partial evaluator even to experimental implementation. Later Carolina Malmkjar from Copenhagen University did it.

Guntis Barzdyn wrote his Ph.D. thesis on the problems of inductive program synthesis. He successfully defended it after Ershov's death in 1988.

## 7. The M2Mix — Dmitry Kochetov

I became the scientific adviser to Dmitry Kochetov by recommendation from Igor V. Pottosin. The subject of his postgraduate research was a partial evaluator for a real-life programming language. Since Dmitry knew very little about mixed computation, he could not even imagine how difficult the task would be that he was going to attack. We chose Modula-2 as the source language. It was not only very popular at that time in the Institute of Informatics Systems. However, Modula-2 was also the language of the programming system that Igor Pottosin developed in the context of a contract with large industrial organization. That raised the chances of practical use of partial evaluator[6].

The M2Mix project required solutions of non-trivial theoretical and implementation problems. For example, since the source language had pointers and arrays, we needed a non-trivial binding time analysis and *alias analysis* in particular.

In order to avoid redundant code duplication, we developed an original *configuration analysis*. Its goal consisted of detecting for each program point the set of variables whose values influence specialization process. Even if the splitting of all program variables into static and dynamic does not change during specialization, the trivial solution that presumes storing and comparing the completely active and reached memory makes such a partial evaluator practically unusable with respect to both time and memory. We can optimize the specialization process by discarding the following variables from consideration:

- variables that do not change in the considered fragment of program. The internal representation of program in the interpreter specialization is an example of such variable;
- variables that can be evaluated based on the values of other essential variables;
- variables not used in the considered fragment, and dead variables in particular[7].

Similarly, it would be very unpractical to compare memory states at each source program point. In order to avoid non-termination it would be sufficient to trace only the set of so-called control points that cuts all program loops that modify static memory. This solution is evidently not optional. The problem is that a small number of control points may lead to residual code duplication, while large number of control points would lead to computational overhead.

Unlike most of the previous projects, we implemented M2Mix as generating extension processor. An important motivation for this decision was efficiency. However, the main reason was more fundamental; such implementation provided an easy way to guarantee that specialization performed exactly the same operations as the ordinary execution. Since we had no other way to execute a program but to translate it by the compiler at hand, we did both specialization and residual program execution in the same way[8].

One of specific features of Modula-2 language is the complete absence of unstructured *goto* control transfer. Being very positive for formal definition of analysis methods, it became a problem when it came to

[6] It should be noted here that partial evaluators were being already used in industrial projects. As an example, one can refer to the works of Samochadin [Samoch82] from Leningrad Polytechnical Institute on specialization of low-level control programs, the works of Romanovsky [Rom95] from the Institute of Automation of SB AS on the optimization of computer graphics programs. But in our case the objective was much more ambitious – to make practically useful a general purpose partial evaluator, which, like a compiler, cannot "know" about a particular application domain, but about the source language only.

[7] The specified conditions are not precise and serve for demonstration of the idea only. For example, a variable may be used in the given fragment, but should be considered essential, since it must be transitively transferred to the fragment where it is used.

[8] Once we spent a lot of time trying to understand why some numerical analysis program and its specialized version produce different results. Finally, it turned out that the problem was in the fact that compiler options that were set for compilation of generating extension differ from those that were set for compilation of residual program.

generation of residual program. To overcome that, we used a well-known trick of modeling labels and jumps by a global loop with nested switch.

Unexpectedly, it turned out that despite the fact that the residual program had minimum of interpretation left, it still worked several times more slowly than the original one. Again, the problem was in the fact that program efficiency depends not only on the number of executed high-level language operations, but also on the compiler's ability to implement them efficiently. In that respect, we could better optimize the compact and clear program of the original interpreter than the residual program with non-local control transfers and intertwined data dependencies. *Post-optimization* focused on this problem. It improved the residual program via common optimizations as well as the special transformations, which take into account the fact that the partial evaluator generated the program.

We tested the M2Mix on the standard test bench, including the Fast Fourier Transformation and the universal scanner Lex. These works formed the basis for Dmitry Kochetov's Ph.D. thesis "Efficient specialization of Algol-like programs". Currently, he works for Microsoft Corporation.

## 8. Binding-Time Improvement

In a certain sense, Nikolay Nepeivoda's prophesy came true. We had one of the most powerful partial evaluators, but had no program on which we could apply it successfully because when a programmer develops a general-purpose program he or she rarely cares about whether it is suitable for specialization or not.

The tools that improve source program binding-time properties would be a good supplement for further enhancement of partial evaluator. Boris Ostrovsky introduced a considerable selection of such transformations. Yuri Bannov in his master thesis has significantly extended this list and in addition, he has formulated conditions for appropriateness of their use. Following Boris Ostrovsky, Yuri Bannov worked on application of mixed computation to parsers, but his freedom of maneuver was restricted to partial evaluator M2Mix and universal parser Yacc[9].

The work titled "Polyvariant binding time analysis for higher-order functions" by Vladimir Ya. Kurlyandchik also focused on binding-time improvement. In contrast to the traditional orientation for the Novisibirsk community toward imperative languages, here the subject of the research was functional programs. The general idea consisted in the following: in usual monovariant binding-time analysis, we declare a variable as dynamic even if there is only one execution when it becomes dynamic. The developed methods allow for transformation of the original program by copying both program and data, so that more computations become static.

The university declared that the master theses of Vladimir Kurlyandchik and Yuri Bannov were the best graduate student's works in the recent years. Now they both work as managers in software companies.

## 9. Postscriptum

In 1992, the Institute of Informatics Systems organized the mixed computation research group. Later, in 1997 they transformed it into the laboratory of mixed computation. The tough 1990s seriously affected the character of research because financial self-provision became the highest priority. Experimental works, whose results could not have immediate use in industrial projects, were either frozen or continued by students. At the same time, computing machinery rapidly changed. New programming paradigms were widely adopted; the computing power increased by several orders of magnitude and they perfected the methods of compilation. Advanced compilers commonly used many features that seemed to be specific to mixed computation such as polyvariancy, loop expansion, and procedure unfolding.

However, it would be wrong to call the experience in the area of specialization without merit. A joint ongoing project between the St. Petersburg company TERCOM and American company Relativity Technologies [Terkom01] is an example. The project focused at developing a means for re-engineering and modernizing legacy applications. One of the key problems there is the *extraction of business logic*. One of the approaches, known as domain-based slicing, is essentially a specialization to known values of program variables in known program points. Even if the method is not suitable for obtaining of object code from an interpreter, it gives a satisfactory solution for many non-trivial tasks that emerge from practice and could unlikely appear in the "academic" setting. For example,

- more than one value or a range of values may be associated with a variable for specialization,

[9] In the course of this work an interesting experiment was carried out that evidently proved the concept's efficiency. The task was to obtain two specialized parser: one should be generated automatically from the universal one, while the other should be written manually from the scratch. Even without mentioning that we could not get rid of all errors in manually written parser, it was simply less efficient than the one generated automatically.

- negative specialization – specification of the set of values to which static variable should *not* be equal,
- specification of the values of static variables not in the beginning of specialized program, but at the point where the values enter the program, e.g. where they are read from the database,
- specialization of programs consisting of many components.

## References


[BarzBul88] Barzdin G.J., Bulyonkov M.A.; Mixed Computation as a Tool for Extracting Compilation Phases, Methods of Compilation and Program Construction, All-Union Conference, Novosibirsk, 1988, pp. 21-23 (in Russian)

[Bul84] Bulyonkov M.A.; Polyvariant mixed computation for analyzer programs, *Acta Informatica*, 1984, Vol. 21, Fasc. 5, pp. 473-484.

[Bul93] Bulyonkov M.A.; Polyvariant binding time analysis, *Proc. of the AC Symposium on partial evaluation and Semantics Based Program Manipulation*, Copenhagen, 1993, pp. 59-65.

[BulErsh86] Bulyonkov M.A., Ershov A.P.; How Do Ad-Hoc Compiler Constructs Appear in Universal Mixed Computation Processes?, *Vychislitel'nye Sistemy*. Novosibirsk, 1986, Vol. 116, Applied Logics, pp. 47-66 (in Russian).

[Ersh84] Ershov A.P.; Mixed Computation, *V Mire Nauki*, 1984, No. 6, pp. 28-42 (in Russian).

[Itkin88] Itkin V.E.; An Algebra and Axiomatization System of Mixed Computation, *Partial Evaluation and Mixed Computation*, North Holland, 1988, pp. 209-224.

[Jones88] Jones N.D.; Challenging Problems in Partial Evaluation and Mixed Computation, *Partial Evaluation and Mixed Computation* (D.Bjørner, A.Ershov, and N.Jones, eds.), Elsevier Science Publishers B.V. *IFIP World Congress Proceedings*, North-Holland, 1988, pp. 1-14.

[Ostr87] Ostrovsky B.N.; Controlled mixed computation and its application to systematic generation of language-oriented parsers, *Programmirovanie*, 1987, Vol. 2, pp.56-67 (in Russian).

[Rом95] Romanovsky A.V.; Mixed Computation application in Computer Graphics Problems; Author's abstract of Ph.D. Thesis, Novosibirsk, 1995 (in Russian).

[Samoch82] Samochadin A.V.; Optimized for Structured Microcomputer Assembler Programs, *Microprocessor programming*, Valgus, Tallinn, 1982, pp. 89-99 (in Russian).

[Terkom01] Terekhov A.A., Terekhov A.N. (eds.); *Automated Program Re-engineering*, St. Petersburg, SPb State University, 2001 (in Russian).