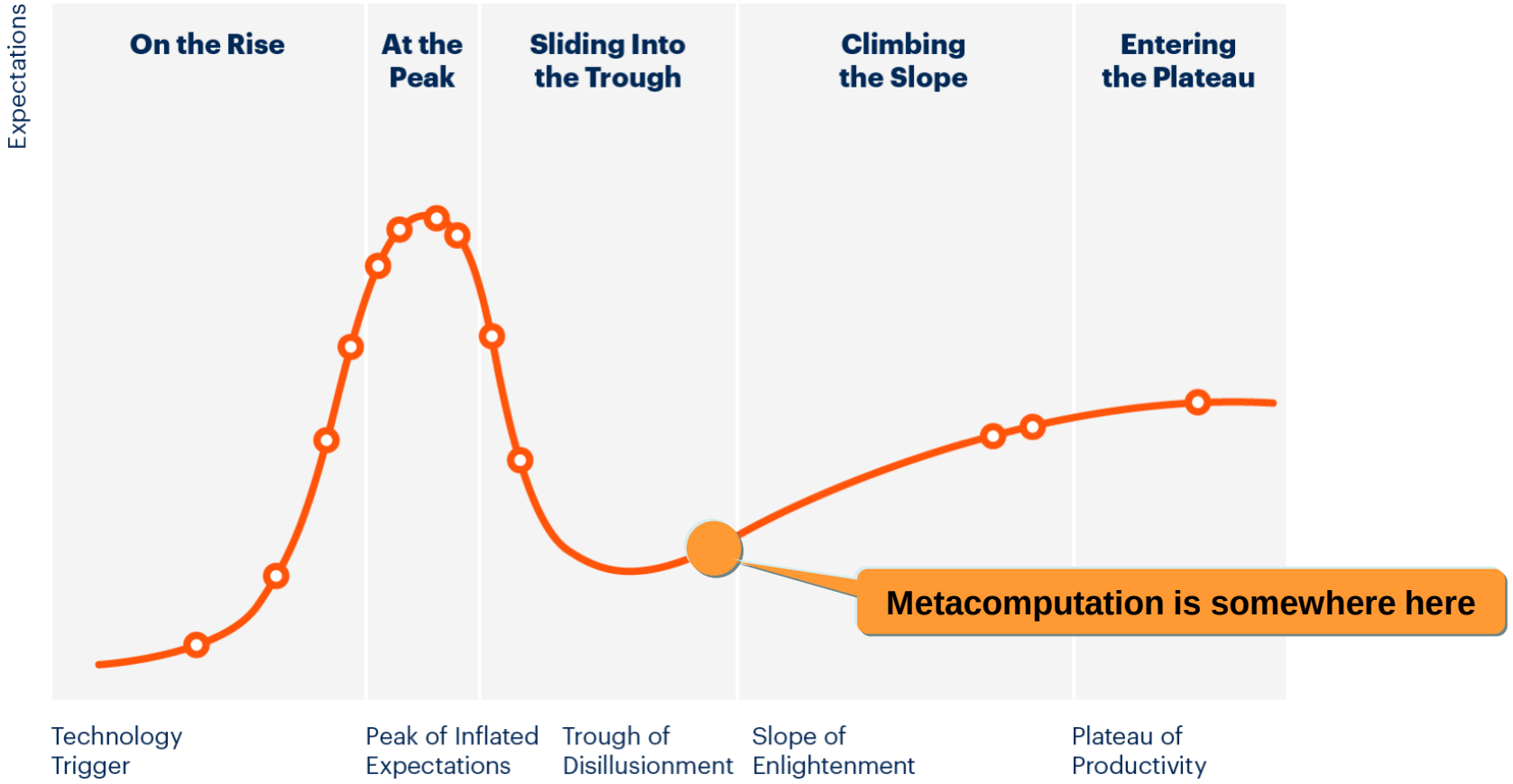
Why are partial evaluation and supercompilation still not widely used in practice? Reflections in light of Russian work on metacomputation

Andrei V. Klimov

Keldysh Institute of Applied Mathematics of Russian Academy of Sciences

Moscow, Russia

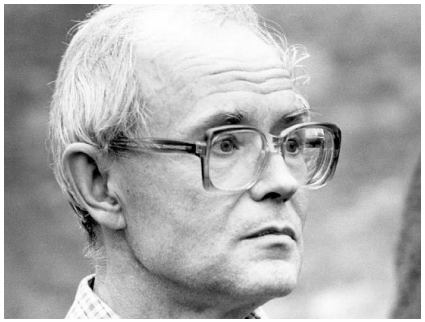
Gartner Hype Cycle



Source (without Metacomputation): https://www.gartner.com/en/marketing/research/hype-cycle

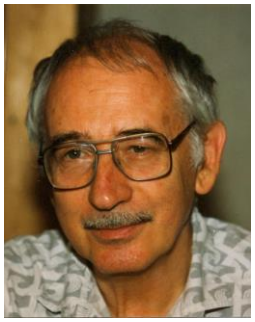
Founders of the area of metacomputation (1970–80s–...)

Andrei Ershov (1931–1988)



Mixed Computation

Valentin Turchin (1931–2010)



Supercompilation

Yoshihiko Futamura

Futamura Projections,
Generalized Partial Computation

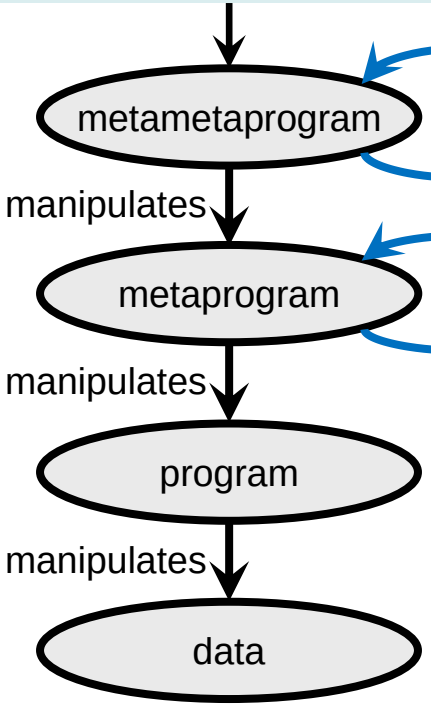
Neil D. Jones

Partial Evaluation,
first spec(spec,spec)

Alberto Pettorossi

Logic Program Transformation
and Verification

Metacomputation



"We must learn how to manipulate computer programs like we manipulate numbers in Fortran."

Valentin Turchin, 1971

A compiler compiler could be produced by specialization of a specializer with respect to itself

3rd Futamura Projection

This is a large-scale **metasystem transition** in terms of the evolution theory by Valentin Turchin:

– Valentin Turchin. *The Phenomenon of Science: A cybernetic approach to human evolution*, 1977.

According to the theory and observations of the general evolution of the world:

– **slow periods of change alternate with rapid transitions leading to a new level of control and the emergence of a next-level metasystem, the growth of the penultimate level**

Based on this, Valentin Turchin expected to see a burst of program analysis and transformations.

– **Has this been happening indeed?**

Program manipulation in practice

Examples of program manipulation

- **Compilers, interpreters** and other language processors if similar kind
- **Various program analyses** used in compilers and in other tools
- **Abstract interpretation**
  - monovariant, polyvariant
- **Program specialization**
  - Partial evaluation
  - Supercompilation
  - Partial deduction
- **Program fusion**
  - Deforestation
  - Supercompilation
  - Partial deduction
- **Program inversion**
  - Supercompilation
  - Partial deduction
- **Program verification**
  - various methods
- *etc*.

- Simply put, according the previous slide, **all this activities are metacomputation**

- However, as is usually the case in general evolution, small changes and the emergence of simple control relate to the level before the next metasystem transition

- Therefore, we call it **metacomputation when program analysis and transformation is "enough" non-trivial, complex and deep**

- The border is rough and approximate, usually revealed by "sudden" growth of complexity:
  - in our subjective opinion, it is somewhere between abstract interpretation and specialization,
  - or in between monovariant and polyvariant abstract interpretation

- The challenges, obstacles and methods in these areas have much in common from a bird's eye view

- Note: We should distinguish:
  - problems and tasks *vs*. methods to solve them

Three levels of obstacles, challenges and approaches to solve

1. **Easy analysis and transformations** with low computational complexity (~linear)
   - Optimizing compilers
   - Working in “black-box” mode without human intervention
   - Obstacle: low complexity, preferably not greater than linear (in practice)
   - *Conclusion: Metacomputation lies beyond this level*

2. **Complex algorithms, almost automatic**
   - Model Checking, SAT-solvers – unexpected success
   - CAD/CAM system for hardware engineering (engines, planes, cars, etc.) with supercomputing
   - Observation: at certain level of hardware/software evolution there is an explosion of applications and rapid development of methods
   - Obstacle: exponential growth of required resources and computer time
   - *Conclusion: Metacomputation tools should use full power of modern supercomputers*

3. **Human-machine systems**
   - A human makes decisions where a machine cannot
     - while a computer guarantees correctness
   - A human knows what is needed, while the machine does not know the goal
     - specifying what is needed is practically impossible
   - Obstacle: the lack of adequate human-machine interfaces, dialogue systems
   - *Conclusion: Metacomputation tools must be in modern IDEs with human-machine interface*

The large-scale MST is already happening and accelerating in the last decade!

What do we observe in practice?

- **Abstract interpretation** and similar program analyses
  - monovariant – widely used
  - polyvariant – rear cases
- **Program generation tools** – great diversity
  - the majority are special purpose ones
  - built-in some languages, e.g., C++ templates
  - recall the surge and decline of macroprocessors in 1960-80s
- **Partial Evaluation** – a lot of interesting theoretical works
  - rear business cases (only recently)
- **Supercompilation**
  - still in research and developments of prototypes
- **Staged computation** – manual separation of binding times
  - present in practice, but how widely is it used?
  - languages: MetaML, MetaOCaml
  - systems: lightweight modular staging
- **Model Checking**
  - very successful method for a particular domain
- **Program verification**, theorem provers, proof assistants
  - rapidly developing during the last decade

In short, **Supercompilation** is a **trace-based** program transformation

1. **Oracle GraalVM with Truffle** Language Implementation Framework with a specializer inside for implementing DSLs by writing an interpreter
2. **Julia language** compiler contains a specializer w.r.to types
3. **AnyDSL compiler framework** for domain-specific libraries (DSLs)

Are there more business cases?

Interestingly, all these specializers use **online Partial Evaluation** without Binding-Time Analysis and with manual annotations and/or appropriate programming style

Selected Western work in metacomputation which mostly influenced work in Russia

1969–1973	**Yoshihiko Futamura**: Partial Evaluation of Computation Process and 3 Futamura Projections (we learned about them about 1980)
1985–1993...	**Neil Jones** *et al*. invented Partial Evaluation with Binding-Time Analysis, evaluated *spec*(*spec*, *spec*) in 1985 and a lot of work and results after
1988–1990s	**Yoshihiko Futamura**: Generalized Partial Computation (~supercompilation with additional information propagation using a theorem prover)
1990	**Philip Wadler**: Deforestation (and later variations by other authors)
1993-1996	**Morten Sørensen**, **Robert Glück**, **Neil Jones**: Positive Supercompilation Morten Sørensen's master thesis: "Turchin's Supercompiler Revisited" with a linear limit
early 1990s	**Valentin Turchin** learned the idea of termination by Kruskal homeomorphic embedding (from the partial deduction community?) and used it in supercompilation
1991–...	**Partial Deduction, Logic Program Specialization** – a lot of people contributed
1990s–...	**Robert Glück** *et al*.: a series of papers on various aspects of metacomputation
2006–...	**Geoff Hamilton** *et al*.: Distillation (~higher-order supercompilation)

4 PhD theses on supercompilation

2001–2002	**Jens Peter Secher**: "Driving in the Jungle" (a paper title, not the thesis)
2007–2008	**Neil Mitchell** (Colin Runciman adv.): "Transformation and Analysis of Functional Programs" (a supercompiler for Haskell)
2007–2008	**Peter Jonsson** (John Nordlander adv.): "Positive Supercompilation for a Higher-Order Call-By-Value Language"
2010–2013	**Maximilian Bolingbroke** (Simon Peyton Jones adv.): "Call-by-need supercompilation" (for Haskell)

Mixed Computation in Novosibirsk, Russia

mid 1970s Andrei Ershov coined the concept of **generative extension** and realized the importance of specialization in a variety of system programming tasks and initiated the development of **mixed computation**

- for imperative languages from the very beginning, working with states

1977–1996 Development of theory of mixed computation and partial evaluation
an attempt to go into practice by implementing a specializer for Modula-2

- Andrei Ershov
- Michael Bulyonkov
- Vladimir Itkin
- Boris Ostrovsky

As I learned from private conversations, they stalled on the problem of side effects and mutable objects.

Supercompilation and partial evaluation in Russia (1)

1974–1975 Valentin Turchin gave a series of seminars on supercompilation with core ideas (driving, configuration analysis, neighborhood analysis, termination) to a group of students in Moscow

1977 **Valentin Turchin** publishes the three “metasystem transition schemes” equivalent to Futamura Projections (metacomputation MST schemes would be later generalized)

1980–1996 A series of papers by **Valentin Turchin** on supercompilation of Refal

- CYNU Report 1980 contains a lot of ideas (underdeveloped till now)

1980s **Valentin Turchin** developed first supercompilers for his functional language Refal (CUNY, NY)

- Supercompilation looks difficult; core notions are not separated well enough

1987–1990 Contributions of **Sergei Romanenko** to results on Partial Evaluation in DIKU

- The invention of PE by Neil Jones *at al*. inspired to look for simpler metacomputation and splitting supercompilation into pieces, besides great achievement of *spec*(*spec*, *spec*)

1990s Research into theory of supercompilation and simplification of supercompilers with DIKU

- **S. Abramov**, **And. Klimov**, **Yu. Klimov** with **N. Jones**, **R. Glück**, **M. Sørensen**, *et al*.

1995 **Sergei Abramov**: book and doctor thesis “Metacomputation and its applications”

1993–2000s **Andrei Nemytykh** continues developing Turchin’s series of supercompilers for Refal

2007 **Andrei Nemytykh**: book and PhD thesis “Supercompiler SCP4: General Structure”

- A lot of interesting experiments with supercompilation
- Proving reachability in Petry Nets (counter systems) by supercompilation

1998–2000s Суперкомпилятор JScp для языка Java (Анд.В. Климов, Арк.В. Климов, А.Б. Шворин)

2002–2009 Partial evaluator CLI-PE for MS CIL.NET (Yuri Klimov, et al.)

Supercompilation and partial evaluation in Russia (1)

1974–1975	Valentin Turchin gave a series of seminars on supercompilation with core ideas (driving, configuration analysis, neighborhood analysis, termination) to a group of students in Moscow
1977	**Valentin Turchin** publishes the three “metasystem transition schemes” equivalent to Futamura Projections (metacomputation MST schemes would be later generalized)
1980–1996	A series of papers by **Valentin Turchin** on supercompilation of Refal – CYNU Report 1980 contains a lot of ideas (underdeveloped till now)
1980s	**Valentin Turchin** developed first supercompilers for his functional language Refal (CUNY, NY) – Supercompilation looks difficult; core notions are not separated well enough
1987–1990	Contributions of **Sergei Romanenko** to results on Partial Evaluation in DIKU – The invention of PE by Neil Jones *at al*. inspired to look for simpler metacomputation and splitting supercompilation into pieces, besides great achievement of *spec*(*spec*, *spec*)
1990s	Research into theory of supercompilation and simplification of supercompilers with DIKU – **S. Abramov**, **And. Klimov**, **Yu. Klimov** with **N. Jones**, **R. Glück**, **M. Sørensen**, *et al*.
1995	**Sergei Abramov**: book and doctor thesis “Metacomputation and its applications”
1993–2000s	**Andrei Nemytykh** continues developing Turchin’s series of supercompilers for Refal
2007	**Andrei Nemytykh**: book and PhD thesis “Supercompiler SCP4: General Structure” – A lot of interesting experiments with supercompilation – Proving reachability in Petry Nets (counter systems) by supercompilation
2010	**Andrei Klimov**: proof that multi-result supercompilation solves a formally defined class of tasks – the first formal characteristic of the power of supercompilation

Supercompilation and partial evaluation in Russia (2)

Theory and supercompiler prototypes

2008–2010 **Ilya Klyuchnikov** (PhD), **Sergei Romanenko**: Multilevel supercompiler for a higher-order language

- termination of a new version homeomorphic embedding for configurations with higher-order terms

2006–2012 Supercompilation relation and domain-specific multi-results supercompilers

- Andrei Klimov, Ilya Klyuchnikov, Sergei Romanenko

2012–2017 **Sergei Grechanik** (PhD): “Proving properties of functional programs by means of equality saturation”

- multi-result supercompilation with equality saturation
- a kind of higher-order supercompilation, but this is not studied yet

Going to practice

1998–2000s **Andrei Klimov**, **Arkady Klimov**, **Artem Shvorin**: Supercompiler JScp for Java (version 1.4)

- (discussion on the next slide)

2002–2009 **Yuri Klimov** (PhD): Specialization of programs in object-oriented languages

- polyvariant partial evaluator CILPE for MS CIL.NET
- further development after Ulrik Schultz (1999–2000)

2017–... **Igor Adamovich**: Partial evaluator JaSpe for Java (discussion on the next-next slide)

Future research directions

- Multi-lever and higher-order supercompilation (including distillation by J. Hamilton)
- Multi-result supercompilation
- Partial evaluation and supercompilation for object-oriented languages
- Synthesis of partial evaluation and supercompilation (~supercompilation in D-part)

Java Supercompiler JScp

- The first version of Java Supercompiler JScp was developed in 1999–2003 at a startup Supercompilers, LLC, and then gradually improved, experimenting with applications
  - Founders: Valentin Turchin, Yuri Mostovoy
  - Main developers: Andrei Klimov, Arkady Klimov, Artem Shvorin
  - Methods are mainly unpublished
  - Several papers with demo problems

- Main observations:
  - A lot of manual control through variety of options is needed to achieve good results
  - Residual code is surprisingly understandable, but it is difficult to capture how it has been produced and difficult to put into correspondence with source code without appropriate tools in IDE
  - It is clear what tools should to be implemented in an IDE to make life easier

Partial Evaluation is best suited for human-machine interaction among specialization methods

- We expect that **Partial Evaluation with Binding-Time Analysis would come into wide practice earlier** than Supercompilation, since its behavior is more understandable for the user due to clear separation of statically evaluated and revisualized code
  - First business cases of PE listed above (Oracle GraalVM, Julia, AnyDSL) do not use BTA
    - psychological reason: maybe, BTA requires some nother way of thinking
    - objective reason: the lack of good tools
  - It seems like just attention and investment from software giants (like Microsoft) are required, but it's surprising that no company has done this for 3 decades

- For practice, **polyvariant** Binding-Time Analysis is required, which is not so easy to visualize

- Polyvariant BTA has much more degrees of freedom, which require user control, than monovariant BTA

- Therefore, a good human-machine interface is needed to use PE in practice

- Screenshot of an example Java program in Eclipse IDE with Java specializer JaSpe plugin from a recent paper by **Igor Adamovich**

```
1  public class MethodExample {
2      @Specialize
3      Number example(double dArg) {
4          boolean S = true;
5          Number obj, res;
6
7          if (S) {
8              obj = new ErshovNumber(1.0);
9          } else {
10             obj = new DummyNumber(dArg);
11         }
12         res = obj.pow(3).pow(2);
13         return res;
14     }
15 }
16 public class Number {
17     double val;
18
19     public Number(double x) {
20         val = x;
21     }
22     @SpecInline
23     Number pow (int n) {
24         return new Number(Math.pow(val, n));
25     }
26 }
```

```
27 public class ErshovNumber extends Number{
28     ErshovNumber(double x) { super(x); }
29
30     @SpecInline
31     Number pow (int n) {
32         double r = 1, x = val;
33         while(n != 0)
34             if (n%2 == 0) {
35                 x = x * x; n = n / 2;
36             } else {
37                 r = r * x; n = n - 1;
38             }
39         return new ErshovNumber(r);
40     }
41 }
42 public class DummyNumber extends Number{
43     DummyNumber(double x) { super(x); }
44 }
```

“Killer” applications and problems to solve

- **Compilers from interpreters** by **Futamura-Turchin Projections**
  - this task does not become a “killer app” as production of compilers is not expensive enough
  - nevertheless: the modern wave of DSLs requires such technologies
  - example: Oracle GraalVM – polyglot VM with Truffle compiler with partial evaluator inside
- **Fusion of components** of applications assembled by **component programming** of various kinds
  - raising demand, especially after Moore's law has slowed down
  - however: modern component programming is parallel and concurrent
  - metacomputation of parallel programs is required (but almost no research!)
- **Compression of hierarchies** of simulation models as a way towards **scalable simulation**
  - to overcome the main obstacle of simulation: diverse granularity of levels of hierarchy
  - simulation resembles interpretation: specialization of a simulator w.r.t. to a model
- **Program verification** is a meta-activity on programs without genuine metacomputation now
  - equivalent transformation to better verifiable form
  - verification by program transformation (to a form with evident answer)
  - example: using supercompilation as normalization in modern proof assistants
- **Artificial Intelligence** really requires a lot of metacomputation tools to go beyond Neural Networks
  - manipulating models of the world (the main omission of the modern AI on Neural Networks)
  - managed, controlled, artificial evolution (considered too slow now, but this is to be changed)

As Conclusion: Return to the three levels

1. **Easy analysis and transformations** with low computational complexity (~linear)
   - Optimizing compilers
   - Working in “black-box” mode without human intervention
   - Obstacle: low complexity, preferably not greater than linear (in practice)
   - **Conclusion: Metacomputation lies beyond this level**

2. **Complex algorithms, almost automatic**
   - Model Checking, SAT-solvers – unexpected success
   - CAD/CAM system for hardware engineering (engines, planes, cars, etc.) with supercomputing
   - Observation: at certain level of hardware/software evolution there is an explosion of applications and rapid development of methods
   - Obstacle: exponential growth of required resources and computer time
   - **Conclusion: Metacomputation tools should use full power of modern supercomputers**

3. **Human-machine systems**
   - A human makes decisions where a machine cannot
     - while a computer guarantees correctness
   - A human knows what is needed, while the machine does not know the goal
     - specifying what is needed is practically impossible
   - Obstacle: the lack of adequate human-machine interfaces, dialogue systems
   - **Conclusion: Metacomputation tools must be in modern IDEs with human-machine interface**

The large-scale MST is already happening and accelerating in the last decade!

Extra slide: Conclusion in Russian
(копия аннотации)

- Основная идея ответа на поставленный вопрос в том, что дело оказалось труднее, чем мечталось отцам-основателям этого научного направления — Валентину Турчину, Андрею Ершову и, наверно, Ёсихико Футамуре и Нилу Джоунсу тоже.

- «Великого оптимизатора программ», работающего нажатием кнопки, не получилось. Первым (общеизвестным) препятствием стала далеко не линейная сложность алгоритмов, не укладывающихся в парадигму использования оптимизирующих компиляторов. Вторым (не столь очевидным) — принципиальная невозможность «перекладывания» этой деятельности на машину. Из причин этого отметим:
  - отсутствие понятия «наиболее оптимизированная программа», к которому могли бы приближаться алгоритмы,
  - главное: отсутствие у машины представлений о целях преобразований, которые варятся в голове у пользователя, а специфицировать непонятно как.

- Тем не менее, подходы к построению систем метавычислений, полезных на практике, существуют, только до сих пор им не уделялось достаточного внимания. Отбросив надежды на легкость задачи, обсудим, как быть дальше:
  - не бояться использовать всю мощность современных параллельных компьютеров и суперкомпьютеров;
  - не бояться алгоритмов, которые требуют такой мощности;
  - главное: строить человеко-машинные системы, диалоговые метавычислительные инструменты, комфортные для пользователей при решении текущих программистских задач;
  - а для этого: реализовывать инструменты для распространенных языков и погружать в привычные интегрированные среды (IDE) и демонстрировать образцы решения задач.

- В качестве материала для выводов используем российский опыт работ по метавычислениям (что отражено в заголовке) и лишь скороговоркой обозначим западные работы, наиболее значимые для нас. У этого две причины:
  - субъективная: хочется назвать малоизвестные работы близких коллег;
  - объективная: из опубликованных западных работ трудно извлечь отрицательный опыт, так как обычно такие результаты не публикуется, а про свои работы мы знаем, куда стремились, что получилось и где споткнулись.