Metasystem Transitions in Computer Science and Mathematics

Draft 2.6

Robert Glück¹ Andrei Klimov²

Institut für Computersprachen Keldysh Institute of Applied Mathematics

University of Technology Vienna Russian Academy of Sciences

Argentinierstr. 8 Miusskaja Sq. 4

A-1040 Vienna, Austria 125047 Moscow, Russia

July 26, 1993

Abstract We analyze MSTs, which may be observed, or are intentionally organized, in computer science and mathematics. These sciences are considered dealing with linguistic modeling. Various metasystems structures are revealed in activities of using the computer as a tool that makes linguistic models behaving by themselves, independently of their creator. The crucial role in automation of creating linguistic models is played by the metasystem transition to metacomputation, that is computation over algorithms. Two metacomputation methods, supercompilation and partial evaluation, are reviewed and compared, and their metasystem structure is exhibited. The problems to be solved by metacomputation are derived from the needs of linguistic modeling: composition, inversion and specialization of programs. The technique of manipulating language definitions and converting compilative definitions to interpretive ones and vice versa, by several MSTs performed by self-application of metacomputation is demonstrated. It is shown how it could be used for compressing hierarchies of mathematical definitions in order to manipulate formal mathematical texts efficiently. The idea of the approach to theorem proving by computer using metacomputation is explained. Finally we review the Cybernetic Foundation of mathematics and discuss its (ultra)metasystem structure.

Keywords linguistic modeling, metacomputation, metasystem transition, program manipulation, supercompilation, partial evaluation, self-application, theorem proving by computer, Cybernetic Foundation of mathematics

Contents

1 Introduction
2 Linguistic modeling and metacomputation
3 Deriving linguistic models by metacomputation
  3.1 Problem of program composition
  3.2 Problem of program inversion
  3.3 Problem of program specialization
4 Methods of metacomputation
  4.1 Supercompilation
  4.2 Partial evaluation
  4.3 Metasystem structure of the methods
5 Manipulating language definitions
  5.1 Converting interpreters to compilers

¹
²
5.2 Converting compilers to interpreters
6 Applications to mathematics
6.1 Compressing hierarchies of mathematical definitions
6.2 Theorem proving by metacomputation
6.3 Basic ideas behind the Cybernetic Foundation of Mathematics
7 Conclusion

1 Introduction

During the last four decades we have been witnessing tremendous technological breakthroughs and the emergence of a new discipline: computer science and technology. The ground for the introduction of a universal, symbol manipulating device such as the computer was well-prepared: formal methods have been introduced during the last centuries in a variety of fields, such as business, engineering and industry (Ershov and Knuth 1981).

But what makes the computer so unique? Before the appearance of the computer, the human created linguistic models (for example, to predict the behavior of physical processes) and had to operate them by hand and mind. Linguistic models don't run by themselves. They have to be operated somehow. Unlike earlier special-purpose machines the computer is a universal device which can run an unlimited number of linguistic models autonomously. Linguistic models that can be executed by a computer are referred to as programs.

A program, being just a linguistic expression built out of pure symbolic material, can themselves become objects of formal analysis and manipulation. This opens the possibility of formal reflection about linguistic models (only mathematics created something comparable: metamathematics, which treats mathematical theories as formal objects). The number of such transitions over linguistic models is potentially unlimited. However, computers were really necessary before one could start to learn more about linguistic modeling on a large scale (human beings are not precise, nor fast enough to carry out any but the simplest procedures). This offers the chance of advancing the scientific method which, after all, is in its essence the creation and use of linguistic models (Turchin 1977).

We refer to processing programs by the computer as metacomputation. The term underlines the fact that this activity is one metasystem level higher than just computation, that is processing of arbitrary data by programs (Fig. 1). A lot of language processors constructed during the forty years of the development of programming and computer science -- interpreters, compilers, macrogenerators, checkers, verifies, automated testing systems, etc. -- fall into this category. However, we consider the area of metacomputation proper to consist of methods of more deep transformation of programs. The purpose of this paper is to demonstrate the problems to be solved by metacomputation and the role of multiple MSTs and ultra-MSTs in solving them. Some of the problems can be already run by computer, others are still open and under current research. It is demonstrated that a lot of problems reduce to three main operations over programs -- specialization, composition and inversion -- and multiple MST by self-applying metacomputation.

The paper is organized as follows. In Section 2 we introduce the basic concepts and notions of linguistic modeling, metacomputation and metasystem transition. In Section 3 the necessity of three metacomputation operations -- composition, inversion and specialization -- in creating linguistic models is demonstrated. In Section 4 two main methods of metacomputation -- supercompilation and partial evaluation -- are reviewed. Section 5 describes how multiple MSTs based on composition and specialization solve the problems of converting compilative and interpretive language definitions into each other. Section 6 is devoted to applications to mathematics: first, the use of converting compilers to interpreters for processing mathematical texts is shown; second, the technique of theorem proving by metacomputation is demonstrated; third, the ideas behind the Cybernetic Foundation of mathematics are reviewed. The material of the paper is based on the ideas and papers by V.Turchin.
2 Linguistic modeling and metacomputation

In this section we discuss the emergence of the computer as a metasystem transition in linguistic modeling. The notion of a model and its relation to an object is reminded and used as a basis for the analysis of the essence of metacomputation. A notation for metasystem transition in the field of metacomputation, which will be used in the subsequent sections, is introduced.

Computer as a tool Science is linguistic modeling of nature. Before appearance of the computer, scientists not only created linguistic models, but executed them by their minds and hands as well. In contrast to models built out of physical material, linguistic models can't work by themselves. To make linguistic models self-acting, to separate them from the author, a special tool was invented -- the computer, which executes models autonomously. Linguistic models of special kind that can be executed by a computer are referred to as algorithms or programs.

The emergence of the computer is an MST with the common structure: before, there were some activities performed manually; afterwards, they has been automated by a new tool. As a result, the amount of linguistic models created and executed has extremely increased (in accordance with the Law of Increasing of the Penultimate Level (Turchin 1977)). {to check the term in "The Phenomenon"}.

However, the computer is an extraordinary tool in the history of technology. It processes not material substances but linguistic models, which are information; to be more specific, discrete information, which posses properties uncommon for material world: it is easily replicable with negligible power expenditures and with the absolute identity (provided computer and other accompanying devices work correctly). It is well-known that the replication is the crucial property in evolution, the property necessary to establish the next level of control, to perform MST. Owning to this, we say that the emergence of the computers starts a new era, The Information Era. We have been witnessing it through fantastic changes in our life and industry, during the last decade especially.

Being a universal device for processing discrete information, the computer is very flexible. It can execute any program, being limited not by its semantics but by the amounts of required memory and time.

The execution of a model is performed by a system of two components: the computer hardware, say M ('machine'), and the algorithmic model, say P ('program'). We denote the action of executing a model P with an initial data x in a machine M producing the result y as follows (to emphasize the computation process, we write out the result after an arrow rather than an equality sign):

\[ \langle P \ x \rangle_M \rightarrow y \]

The program P is a text in the language understood by the machine M. We refer to this language as M as well. Unless we speak about several different machines and languages, we shall skip the index denoting the machine:

\[ \langle P \ x \rangle \rightarrow y \]

For uniformity, we shall use the notation \( \langle P \ x \rangle \) for a function call (in the common mathematical sense) as well, even if it is not considered running a machine, because usually our next step is defining the program evaluating the function P.

```
+-------------------+       +-------------------+
|       Program     |       |     Computer      |
+-------------------+       +-------------------+
      |     +-------------------+       |     +-------------------+
      |     |        |     +-------------------+       |     +-------------------+
      |     |  instructs   |     |     executes    |     |     |
      |     +---------------+       +---------------+       +---------------+   
      | Computer            |       |     Program      |
      +-------------------+       +-------------------+
```

Fig. 2: Two views at the computer and the program, two metasystem structures.

Two views at the computer The system of a computer and a program can be considered a metasystem by two ways (Fig. 2). From the one viewpoint, "a program controls, instructs a computer"; from the other, "a computer executes a program". In the former viewpoint, a program is a control subsystem and a computer is a subject system to be controlled, the latter is quite the reverse.
It is interesting to note that in the early years of programming, the first viewpoint dominated in programmers' minds. It can be still found in constructing of real-time and embedded systems. The computer, not the program, was in the focus. In 60s--70s it was fashionable to note at the lectures on "advanced software technology" that while constructing a program, one should think about the program itself, and that its logic and even the beauty should be of main concern rather than the "efficiency" of its running in the computer. This shift in minds correlated with the general process of program becoming an object. The Metacomputation Era started.

The program is a subject, a controlled level, of two MST hierarchies: software and hardware (Fig. 3). Having different nature, hardware and software develop to a large extent independently, though use the achievements of each other. Each of them makes its own contribution into the evolution of methods of linguistic modeling in general. However, whereas the progress in hardware technology produces quantitative results (such as increasing processor speed and enlarging computer memory), the potential of the software evolution is qualitative. Just the later is of concern to us. We are especially interested in series of MSTs, which form ultrametasystem transitions (Turchin 1977).

Fig. 3: Two metasystem hierarchies over programs.

Dimensions for MSTs in linguistic modeling The linguistic modeling is not just the execution of models; it includes the following activities:

*1. Creating a model of an object.

2. Putting forward a question to be answered by modeling.

*3. Executing the model.

4. Using the result of modeling as a prediction about the object.

As far as the evolution of the methods of linguistic modeling proper are concerned, only the first and third activities, marked by asterisk, are of interest, since the others depend on the external goals of the model user. Thus, two dimensions for MSTs in modeling are under question (in both cases, control means the automation of the respective activity):

control of executing models = computation

control of creating models = metacomputation

The term metacomputation may be defined in two ways. The first is just to say that it is some next level of control over computation. The second, presented by the formula above, is to associate it with the activity of creating models. It is not evident
in advance that they are instances of the same MST. So, let us firstly consider them separately.

Control of executing models The creation of the computer is the first step in controlling the execution of models. However, it is not the last step. It is common for MSTs to occur in series. Though evolution is blind (except when the human will directs it), such series may be considered achieving a goal. The goal is often characterized as performing some existing activity quicker, better, etc. The first MST creates something minimal that is stable, replicable and makes first gains in achieving the goal. After subsequent MSTs achieving the goal will be done 'better'. While discussing the first MST from a series, it is important to understand what is achieved already and what is not achieved yet and remains for subsequent MSTs.

Not all linguistic models can run in the computer. There are limitations of theoretical and practical nature. Firstly, the majority of linguistic models cannot be executed by a computer in principle. For example, one of the most important questions about computations cannot be answered algorithmically: whether a given process will stop or not. Essentially the same property holds for mathematical theories: any theory that is no less powerful than arithmetic, contains a statement which can't be neither proven, nor refuted (though it is semantically true). It is a general property of linguistic modeling that formal languages allow to describe more situations, to put more questions, than they can answer.

Secondly, not any model can be executed in practice, though it defines a precise sequence of actions for the computer. The point is that the execution of models requires time and space in the computer. In many cases, any realistic resources are insufficient.

Both limitations raise the question of building the next levels of control of executing algorithmic models. They are aimed at evaluating algorithms that were theoretically non-executable before, and to execute executable ones using less resources. To achieve this, only two ways exist (except using a more powerful computer, which is evidently a partial solution): either to recast a program into such a form in which it can be executed well, or to automate the process of creating programs at early stages when a program does not exist yet. The former way requires developing technique of transforming programs. Since a program text is an ordinary data, which can be submitted to the computer, it is natural to use the computer itself for transforming algorithms. Thus, a new kind of computation appears -- computation that transforms algorithms. This is metacomputation in the first sense out of introduced above. In terms of modeling, metacomputation is building another model from an existing one. So, we come to the topic of creating models. The later way, controlling the process of creating programs at early stages, is the topic of the next subsection.

Control of creating models The automation of executing models was primarily achieved by the creation of the computer; but the automation of creating models was not affected just by its existence. At the beginning, it was fully performed by the human. The human activity of creating models in a form suitable for executing by the computer is programming.

For more than four decades computer science has been developing methods for the simplification and automation of programming (using the computer itself, of course). Many approaches have been tried such as the development of new language paradigms (e.g. logic, functional, object-oriented, etc.), tools for automating the construction of compilers (e.g. scanner and parser generator, compiler-compilers), various methods for program verification and transformation. However, the problems still exist and the technological breakthroughs on the hardware side have not solved the problems with the creation of software. A situation which has been aptly described as 'high-technology basket weaving'.

A paradox situation: while software has been created with the goal to solve problems more efficiently with less human effort and intervention, one has not yet been able to cope with the problem of software development in an satisfactory manner. However, this mirrors the fact that the control of executing models (the first MST) was directly achieved by the introduction of the computer, while the reasonable control of creating models (the second MST) is not achieved yet.

So, we come to two related problems: how to automate the creation of linguistic models, in general, and that of programs, in particular.

As far as linguistic modeling is concerned, we limit ourselves to using for this task just linguistic modeling itself. We shall not deal with other methods, such as, for example, training in different human-oriented techniques named 'art of computer programming', such as 'structured programming' or other programming paradigms; or psychological methods of 'making inventions', 'brain-storming', concentrating one's mind on a subject, etc. However, our decision does not mean that we argue against these methods. The reason is the intention to achieve self-application and, hence, to build a world in which a series of MSTs of similar nature, that is an ultraMST, is a usual thing.

Moreover, the methods under question will be applied when the desired model is already somehow defined as a linguistic one, though, maybe, in a form unsuitable for some reasons. Therefore, we shall deal with automation of creating linguistic models as just transforming them. This means that we consider the world of linguistic models closed. The act of creating first models is beyond our consideration: it is a creative step of the user of linguistic modeling. On the other hand, the world of programs is open into the wider world of linguistic models: if a program is somehow formally specified, that is its linguistic model is given, its specification becomes subject of transformation aimed at derivation the efficient program.
Nevertheless, the essence of the metacomputation approach is considering algorithmic models as basic material for linguistic modeling, because algorithms can run autonomously. Only when non-algorithmic specifications are the only possible ones, we deal with them. Starting with algorithmic models, metacomputation will expand over the whole world of linguistic modeling, as far as methods of algorithmic transformation of non-algorithmic specifications will develop.

Before going further, let us summarize the ideas behind the notion of a model.

**Modeling** Assume we have an object, say \( o \), whose initial state is characterized by some information, say \( x_o \). Suppose we want to know some fact, \( y_o \), about its behavior, future states, etc., that is, to make the prediction \( y_o \) about the object.

Sometimes the fact \( y_o \) can be determined directly from the object. Then it may be remembered for making predictions about the object in similar situations. Another method, **modeling**, is using a different object \( m \), considering it 'similar' to the original object \( o \). The second object \( m \) is referred to as a **model** of the first one. The model may look nothing like the original; it may even be not a material object but the behavior, functioning of some other object (this is just the case in computer modeling.) The emergence of modeling is a large-scale MST. (The discussion of this is beyond the scope of this paper.) In the final paper, this topic will be discussed in more detail if it is not concerned in other papers of the issue.

![Fig. 4: Object and model.](http://209.85.135.132/search?q=cache:otcXJE0oI1JwJ:ftp://ftp.vub.ac.b...)

In Fig. 4 the correspondence between an object and its model is denoted by the mappings, \( H_x \) and \( H_y \), from the information \( x_o \) and \( y_o \) about the object \( o \) to information \( x_m \) and \( y_m \) about the model \( m \). A model contains less information than the object, just the information we consider 'relevant'. Having the full information about the object, we can deduce the corresponding information about the model by the mappings \( H_x \) or \( H_y \), but not vice versa. Such mappings are supposed to preserve certain properties of the object and are referred to as **homomorphisms**. Having information about the model, \( x_n \), we predict not a precise fact, \( y_o \), about the object but only that \( \langle H_y, y_o \rangle \rightarrow y_n \), that is that a possible state belongs to the set \( \{ y_o \mid \langle H_y, y_o \rangle \rightarrow y_n \} \).

**Creating models from models** To speak about manipulating models, we are to concretize the material which our models are built of. So, from here on, we shall limit ourselves to linguistic models, which are expressions in some language.

Let the object \( o \) be a linguistic model and assume that we want to build another model \( m \) being a model of the first model \( o \). So, we are provided with the object \( o \): a description of the sets \( X_o \) and \( Y_o \) over which \( x_o \) and \( y_o \) range, and with a description of a function \( F_o : X_o \rightarrow Y_o \). We need to define sets \( X_n \) and \( Y_n \) which \( x_n \) and \( y_n \) range over, and the function \( F_n : X_n \rightarrow Y_n \). What can be done manually and what can be automated?

The choice of the information used for building the model is a creative step depending on external goals; by deciding it, we choose what information is to be abstracted and represented in the model. Furthermore, it is natural to write out the homomorphisms \( H_x \) and \( H_y \) simultaneously with designing the material of the model. So, let them be left to human being. However, the difficult task is constructing the function of the new model, \( F_n \). From Fig. 4 we see that \( F_n \) can be determined by using \( H_x \) and \( H_y \) (where \( H_x^{-1} \) is the inverse homomorphism with respect to \( H_x \)).

\[
(*) \quad \text{def } \langle F_n x_n \rangle = \langle H_y F_o \langle H_x^{-1} x_o \rangle \rangle
\]

The full information about \( F_n \) contains in \( F_n, H_x, \) and \( H_y \) and is represented by the formula above. To derive an algorithm from it, two problems that are seen from the structure of the formula must be solved. The most severe one is executing an inverse
homomorphism, the $H_x^{-1}$ (even the meaning of the formula is to be adjusted because generally there are many $\chi_\circ$'s that may be considered the values of $\langle H_x^{-1}, \chi_\circ \rangle$). The second problem concerns efficiently evaluating the composition of functions. We discuss the two problems in the next section in turn, but beforehand, let us introduce the notation for metacomputation.

**Notation for metacomputation** As seen from the foregoing, the function of a new model can be defined by some formula, e.g. by $\langle \ast \rangle$. The formula contains free variables ($\chi_\circ$ in our case). These variables range over initial information with which the new model is to be executed. Though the formula fully specifies the new model, it may be not efficient enough as instructions for the computer or even non-executable practically (like the inverse homomorphism). Therefore, we need some transformation of the model defined by a formula into an efficient algorithm. This process is just metacomputation. Let us denote it by $M_C$. It is applied to a formula proper (not to its value); so, we need some special notation for such an application. In order not to confuse this with the application to the value of the embedded formula, we move the subject formula downwards (filling the remaining place by a line). The metacomputation of our example is then written out as follows:

$$
\langle \mathcal{M}_C \quad \vdash \quad \langle H_x \circ F \circ H_x^{-1} \chi_\circ \rangle \quad \Rightarrow \quad F_n \rangle
$$

We refer to such a formula as an **MST-formula** because it describes the activity of creating models, which is a meta-activity as compared with just executing them. Here $M_C$ is the control subsystem of a metasystem, the formula $\langle H_x \circ F \circ H_x^{-1} \chi_\circ \rangle$ is a subject subsystem. The formula closely correspond to the MST schema in Fig. 5.

---

**Fig. 5: Metacomputation as a metasystem.**

Pay special attention to that free variables occurring in a subject formula have become bound, i.e., that they are not to be replaced by values before executing $M_C$, but stand for themselves; they are just linguistic objects. Moving down is actually quoting; the following may be considered equivalent to the previous one:

$$
\langle \mathcal{M}_C \quad \vdash \quad \langle F(A, \chi) \rangle \quad \Rightarrow \quad F_n \rangle
$$

The two-dimensional notation is much more readable than the quoting, especially in the case when a free variable is to be incorporated in it. For example, consider metaevaluating the formula $\langle F(A, \chi) \rangle$ where $A$ is a constant and $\chi$ is a variable:

$$
\langle \mathcal{M}_C \quad \vdash \quad \langle F(A, \chi) \rangle \quad \Rightarrow \quad F_n \rangle
$$

Suppose we want to replace the linguistic object $A$ by a variable $a$ that is free in the whole MST-formula. We write out this case as follows (the bullet $\ast$ is a place holder denoting the position where the value bound to $a$ is to be inserted):

$$
\langle \mathcal{M}_C \quad \vdash \quad \langle F(\ast, \chi) \rangle \quad \Rightarrow \quad F_n \rangle
$$

Such a formula may become a subject of metacomputation itself. This can happen several times. In such a manner, the formulas of the following form emerge:
Such a formula should be read as follows: The variables of the top level, $x_1', y_1', \ldots$, are free variables of the whole formula. When they are given values and the formula is evaluated, that is, the topmost $Mc$ runs, the result produced is a program whose arguments are the variables of the next level, $x_2', y_2', \ldots$, and whose function is equivalent to the formula under the top level. When $x_2', y_2', \ldots$ are given values and the program obtained at the previous step runs, the next program is produced whose arguments are the variables of the next level, $x_3', y_3', \ldots$ and so on till the bottom.

Two-dimensional MST formulas and the rule of reading them was suggested by V. Turchin. However, in early publications, e.g. in (Turchin, Klimov et al. 1977), one-dimensional notation was used.

3 Deriving linguistic models by metacomputation

In this section, basing on the general schema of linguistic modeling introduced in the previous section, we consider typical problems in computer science to be solved by metacomputation: program composition, inversion and specialization. The metacomputation methods which are capable of solving these problems are discussed in the Section 4 below.

3.1 Problem of program composition

Let us firstly consider the case of abstracting a new model from an existing one without such a complex construct as an inverse homomorphism. This is the case when initial data for the source model, $x_n'$, coincides with the initial data for the new model, $x_n$, that is, the homomorphism $H_x$ is identical (Fig. 6). The reduced model is defined as follows:

$$\text{def } \langle F_n \times \rangle = \langle H \langle F_o \times \rangle \rangle$$

![Diagram] (Fig. 6: Deriving a model by abstracting only output information)

A demonstrative example of such model abstraction is provided by CAD (computer aided design) systems for designing integrated-circuit chips. The logic (that is, the model) of a chip is described by a very large graph composed of thousands of function elements ranging from such simple ones as and, or, delay, to rather complex elements. Arcs connect the inputs and outputs of elements; some of them are the input and output lines of the whole chip. The function of a chip, $F_o$, takes the values of signals in the input lines, $x$, and computes all signals in the inner and output lines, $y_o$. While designing and debugging a chip, an engineer has to perform a lot of test runs. To compute signals in all lines may require such great computer resources that practical chips can not be simulated directly. However, a designer is usually interested only in signals of a rather small part of lines, about several dozens of them. It is naturally to expect that because only a partial task is given, the model of the chip may be drastically reduced. To define the model abstracted in such a way, we are to write out a simple homomorphism $H_y$ that just picks out of thousands of signals only some dozens of them.
However, being evaluated directly, the formula $\langle H_y, F_n, x \rangle$ does not decrease the amount of computer resources needed to obtain the result: as before, at first the set of all signals, $\langle F_o, x \rangle$, is generated and only afterwards the required signals are picked out by applying $H_y$. To earn a profit, we metaevaluate the new model:

$$\langle Mc \text{ ~__________} \rangle \rightarrow F_n$$

$$\langle H_y, F_n, x \rangle$$

It would appear reasonable that in $F_n$, as compared with $F_o$, all redundant computations are thrown away, and actually needed computations are folded into a compact algorithm.

Another application of metaevaluating the composition of functions will be discussed in detail in Section 5.1: deriving an efficient interpreter of a language $\mathcal{L}$ from an interpreter of another language $\mathcal{L}'$ and a compiler from $\mathcal{L}$ to $\mathcal{L}'$. Here is just its MST schema:

$$\langle Mc \text{ ~__________} \rangle \rightarrow \text{IntN}$$

$$\langle \text{IntL, } \text{CompNL, } p \rangle, x \rangle$$

One more application will be presented in Section 6.2 on theorem proving. It will be especially interesting that the basic model, $F_o$, will be the $\mathcal{H}_C$ itself; hence, it will be the case of self-application.

The importance of successful metacomputation of the composition of algorithms is difficult to overestimate. Composition of functions is the main method of constructing complex objects from simple ones in mathematics. It is so basic and natural that we often do not notice when we actually use it. The problem arises that such compositions are to be flattened.

### 3.2 Problem of program inversion

The majority of mathematical problems are stated in the following way: provided the description of the properties of an object, $F$, is given, find (at least one) such an object $x$ that $F(x)$ is true. It is referred to as an inverse problem. The predicate $P$ can be formulated as an algorithm checking the linguistic object $\gamma$, though often it is non-executable practically. In time, more and more non-executable algorithms will become executable due to maturation of the metacomputation methods. However, many problems are already stated using executable check algorithms. They range from rather simple ones to such huge as the CAD/CAM example discussed above.

Let us consider it once more. In computer aided manufacturing, a very important problem is the test generation. It is actually an inverse problem: having the model of a chip, $F_n$, that evaluates signals in some output lines, $\gamma_n$, from input signals, $\zeta$, an engineer is interested in finding such a set $\{x_1, x_2, \ldots, x_n\}$ that $\gamma_n$ equals to each of $\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ respectively:

$$\langle F_n, x \rangle = \gamma_n$$

$$\langle F_n^{-1}, x \rangle = \gamma_n$$

Several efficient methods of deriving $F_n^{-1}$ from $F_n$ were suggested and are under further investigation now (Turchin 1972) (Romanenko 1988) (Romanenko 1991) \cite{1}. There exists a very simple one, which is often considered a non-solution, because of being impractical. However, keeping in mind metacomputation, such an algorithm has sense as well. It takes the definition of the algorithm $F_n$ and the result value $\gamma$ and computes the argument $\zeta$ in the following way: looking through the whole set of $\zeta$'s, execute $F_\zeta$ for each $\zeta$ and return first $\zeta$ such that $F_\zeta$ equals to $\gamma$. Let us denote this algorithm $\langle \text{SearchArgF} y \rangle$.

**Inverse problem solver as a metasystem** The algorithm above is actually a metasystem over $F_n$. It takes a description of $F_n$ as an argument and performs some special form of control of its execution. Let us express it in form of MST-formula. Let Search be a process that takes a formula with one free variable and, binding the variable to each linguistic object in turn, evaluates the formula until the object is found, for which the formula is true. Then we can write out the simplest inverse algorithm as follows (f is lower case to denote a free variable in the function definition):

$$\text{def } \langle \text{SearchArg f} y \rangle = \langle \text{Search f y} \rangle$$

where $\langle \text{Equal a, b} \rangle$ is the equality predicate.

**The second MST** Given an algorithm solving the inverse problem, e.g. SearchArg, we can metaevaluate it to try to produce an executable one:
The performance of the result algorithm, $F^{-1}$, depends upon the complexity of the source algorithm, $F$, the power of metaevaluator, $M_C$, and the particular inverse problem solver, $\text{SearchArg}$.

The last formula is the second MST over $F$, the first MST being $\text{Search}$. To exhibit two MSTs clearly, let us substitute the $\text{SearchArg}$ definition into the formula above:

\[
\begin{align*}
\text{def} & \quad F_n \rightarrow F_n^{-1} \\
\langle \text{SearchArg} \ x \ y \rangle & \rightarrow F_n^{-1} \\
\langle \text{SearchArg} \ n \ y \rangle & \rightarrow \langle \text{Equal} \ x \ y \rangle
\end{align*}
\]

The third MST can be performed as well. However, we postpone discussing it till Section 5 where the role of several MSTs in generating interpreters from compilers, and vice versa, is demonstrated.

**Abstracting a model using inverse homomorphism** Let us now consider the second special case of abstracting a new model from an existing one, when only an inverse homomorphism is used. This is the case when output from the source model, $\circ$, coincides with output from the new model, $\circ_n$, that is, the homomorphism $H_{\circ}$ is identical (Fig. 7).

![Fig. 7: Deriving a model by abstracting only input information](image)

The function of the new model, $F_n$, is here defined as follows:

\[
\text{def} \quad F_n = F_n^{-1} \circ F_n \langle H_{\circ}^{-1} \ x \ n \rangle
\]

Like above, to build a more efficient model $F_n$, we perform MST by metaevaluating the definition above:

\[
\begin{align*}
\text{def} & \quad F_n \rightarrow F_n^{-1} \\
\langle M_C \ F_n \ Y \rangle & \rightarrow \langle F_n \ F_n^{-1} \ x \ n \rangle
\end{align*}
\]

Here the simplest combination of the two methods of composition and inversion is used.

What is the sense of this case? We want to build a new model that can be used to make a prediction $\gamma$ using partial information $\gamma_0$ about the input data $\gamma_0$. What is known about $\gamma_0$ is defined by the homomorphism $H_{\circ}$. For example, ...{CAD/CAM}...

Here are two opportunities. In some instances, the partial information $\gamma_0$ is sufficient to produce a concrete $\gamma$. Then $F_n \ x_n$ should return it. However, generally $\gamma_0$ does not define output $\gamma$ precisely. In this case we can be interested either in any possible output $\gamma$, or in the set of all possible outputs generated one by one. The representation of the output set may be different
and is determined by the method of constructing inverse algorithm $H_x^{-1}$.

So, we see that varying the method of solving an inverse problem, like SearchArg, and metaevaluator $M_C$ different applications can be generated by MST-formulas involving composition, inversion and multiple MSTs.

### 3.3 Problem of program specialization

Are there problems of metacomputation that are simpler than composition and inversion but still important? The work by Y. Futamura (Futamura 1971) has given a positive answer to the question. The solution to the problem of generating compilers from interpreter found by Y. Futamura (discussed in Section 5.1 below) uses metacomputation of neither composition of programs, nor inversion. The problem was reduced to so called specialization.

![Fig. 8: Specializing a model](http://209.85.135.132/search?q=cache:otcXJE0o1JwJ:ftp://ftp.vub.ac.b...)

The problem of specialization of a model arises when the problem space to which it is applied is narrowed. This means that the function of a new model, $F_n$, is the same as that of original, $F_o$, but the argument variable(s), $x$, ranges over a part of input values:

$$
\text{def } F_n(x) = F_o(x) \quad \text{if } x \text{ belongs to } S, \text{ where } S \text{ is a subset of the domain of the model } o
$$

The reason to metaevaluate the narrowed model is that the definition of the original model may contain redundant information that does not apply to the subset domain. Processing this information while running the model, is an extra work. So, the narrowed model may be optimized.

**Particular cases of the problem of specialization** In fact, the problem of program specialization reduces to the problem of program composition. So, it is of interest only if simpler methods for solving it have been invented. Let us consider the complexity classification of its special cases (Fig. 8). The classification reflects the state of art of the metacomputation methods, which are reviewed in Section 4 below.

1 **The general case** The above problem statement of specialization is the most general one:

$$
\text{def } F_n(x) = F_o(x) \quad \text{if } x \text{ belongs to } S
$$

Because of its generality, no better methods than those used for program composition have been found till now. The question is to choose the means to represent and manipulate the description of $S$. To allow all computable subsets in the role of $S$, we are forced to deal with algorithmic descriptions of $S$ of the general form. In this situation, following Occam's razor, in order not to create more entities than needed, the best solution is to reduce the problem of specialization to program composition as immediately as possible. The most direct reduction, which is in good agreement with the supercompilation methods, was suggested by V. Turchin. It is as follows. A set $S$ is represented by a program $F_S$ (called a filter) that describes the function that is identical on the set $S$ and undefined on other values (that is, the program raises an error):

$$
\text{def } F_S(x) = x \quad \text{if } x \text{ belongs to } S
$$

The specialized model is then defined by the composition of the original model $F_o$ and the function $F_S$: 

```

def \langle F_n \ x \rangle - \langle F_o \ F_S \ x \rangle

Note that the composition of \( F_o \) and \( F_S \) in the reverse order:

\[
\def \langle F_n \ x, y, z \rangle - \langle F_o \ F_S \ x, y, z \rangle \\
\def \langle F_s \ x, y, z \rangle - \ldots x \ldots y \ldots z \ldots x \ldots 
\]
defines specialization of the model \( F_o \) as well, but now its range is narrowed, not the domain. This case occurs when we are interested only in those results of modeling that belong to a given subrange, and we would be satisfied by the output "an error" if an input that evaluates to output beyond the limits is supplied.

So, a particular success in solving the problem of specialization should be expected only if some limited classes of subsets are considered. This is just what actually has happened. Of the following special cases, the first one is dealt with by supercompilation, the second -- by partial evaluation.

2 Expressions with free variables From the MST-formulas above (as well as below) one may see that what is primarily needed is to manipulate sets represented by linguistic expressions with free variables. Fortunately, this is not just a primary need, but also a basic constructive means to deal with sets.

Reducing to composition, this case is expressed by a set-producing function with the right-hand side being a passive expression without function calls, just with the variables defined in the left-hand side:

\[
\def \langle F_n \ x, y, z \rangle - \langle F_o \ F_S \ x, y, z \rangle \\
\def \langle F_s \ x, y, z \rangle - \ldots x \ldots y \ldots z \ldots x \ldots 
\]

Amongst MST-formulas, that of the following kind describe this case of the problem of specialization:

\[
\langle MC \ldots \rangle \\
\langle F_\ldots x_\ldots, y_\ldots, z_\ldots \rangle - \text{just one call} \\
\langle \ldots \rangle - \text{arbitrary expression} 
\]

Operations with sets represented by expressions with variables form the basis of supercompilation.

3 Part of arguments known However, even more special case is most elaborated now due to its simplicity. If the input to a model is split into several arguments, they can be supplied to the model in turn. For example, if the model has two arguments, \( \langle F_n \ x, y \rangle \), and is run many times with the same first argument \( x = \lambda \) and with the second argument \( y \) varied, the specialized model with the first argument fixed may be constructed to increase efficiency:

\[
\def \langle F_n \ y \rangle - \langle F_o \ \lambda, y \rangle \\
\def \langle F_n \ x, y, z \rangle - \langle F_o \ F_S \ x, y, z \rangle \\
\def \langle F_s \ x, y, z \rangle - \lambda, x, y, \beta, z, \gamma 
\]

Unexpectedly many problems were found to reduce to this particular case, including the central problem of metacomputation -- the problem of self-application (Futamura 1971). During last decade this case was studied most intensively and the development of the group of methods named partial evaluation has been actually completed (Jones, Gomard et al. 1993).
4 Methods of metacomputation

In this section, we review two methods of metacomputation: supercompilation and partial evaluation. The former is rather general and is aimed at solving all the problems discussed in the previous section: composition, inversion and specialization. The later performs only specialization, but is significantly simpler. Owing to this, it was the first to be self-applied. (The self-application is discussed in the Section 5 below.) In the last subsection 4.3 we discuss the metasystem structure of both methods and compare them.

The knowledge of the methods is not needed for understanding the other sections of the paper. The reader may skip this section.

4.1 Supercompilation

The metacomputation method that could solve the problems above was suggested by V.Turchin in the early 70s and has been under further development by him and other researches till now. It is referred to as supercompilation (Turchin 1972), (Turchin 1980), (Turchin 1986), (Turchin 1988), (Abramov 1991), (Glück and Klimov 1992).

Ordinary computation is a linear process of steps that are picked out from a program due to given data. A set of processes picked out by a set of input data is an obvious MST (Fig. 10). To deal with a set of processes, one need to perform (meta)computation with sets of states instead of concrete states. Linguistic modeling suggests a basic mechanism of describing sets: by expressions with free variables (such as that we write under the line in MST formulas). Of course, not all sets regarded by the set theory can be represented in such a way. Yet we need not this. We could behave in such a way: firstly, consider the basic sets, and only when they are not sufficient, invent more complex constructs, preferring using already constructed processes, composing some MST figures of them.

Having chosen basic linguistic material to represent sets of states -- expressions with variables -- the algorithm of their processing is to be constructed. It was developed by V.Turchin (the first publication is (Turchin 1972), the last version is (Turchin 1986)) and is referred to as driving. It is just generalization of ordinary computation with concrete data to the case of sets of data. In particular, given an input set consisting of just one state, the driving produces one ordinary process.

In general, driving works as follows. It tries to perform a step and if a free variable does not hinder it, the step is done. Otherwise, it means that the conditional can't be evaluated because of variables in data met by the program. Then the current set of states is divided into two parts: one consists of states at which the conditional is true, and false at another. Both subsets are represented as next nodes. So, during metacomputation a tree is developed instead of a plain sequence of states built by ordinary computation.
The construction of the result program is not completed by building the tree representing the set of processes, though the tree contains the full information how to apply the program to data, and therefor it can be used as a program. The problem is that the tree is potentially infinite in general case, but the result program must be a finite text. If the infinite process tree could be folded into a finite graph without losing the information how to apply it, the problem is solved. Note, that if the set of states in some of the nodes (e.g., node 2 in Fig. 11) of the process tree equals to one of the previous ones (e.g., 1), the trees developed from both nodes will be equivalent, so we need not develop the node 2 and may just record the reference to the node 1. It means that when we need to use the node 1 we can take the node 1 instead. (Interpreting a process graph as a program, this is just a goto operator.) We refer to such an action as looping-back. If occasionally all infinite processes are looped-back in such a way, the finite graph is obtained, which is the representation of the result program.

![Fig. 11: Looping-back 2 to 1, 4 to 3](x's are terminal nodes)

However, we will not be so lucky in general case: the majority of process trees has infinitely many different nodes. In this case we can approximate the process graph: without losing the information about how to apply it, a node may be replaced by some node representing a larger set of states. We refer to this action as generalization. What is lost is the quality, "efficiently" of the result program, because now some useless walks that correspond to no actual processes, exist in the graph. More generalizations are performed, worse the result program. However, generalization is inevitable payment for the finiteness of the result graph.

Summing-up, the method of supercompilation consists of the following:

1. **Driving** that interprets a set of states and constructs a potentially infinite process tree.

2. **Looping-back** a node in a tree to an equivalent predecessor.

3. **Generalization**, which performed in a proper way can guarantee that all infinite processes are looped-back.

Driving is an immediate MST over computation: a transition from computation of a concrete data to (meta)computation of a set of data. Looping-back is a means of folding an infinite process tree to a finite process graph, which is the representation of the result program. Generalization is the most complex operation in supercompilation: in essence, it is building a model of a set of processes by forgetting the precise information about the set. Being a case of a rather complex phenomenon -- linguistic modeling, generalization can't be implemented by one method for all times. Some methods of finding reasonable generalizations are already constructed and used, others are under further investigation. While developing generalization methods, more MSTs are performed as well. However, it is beyond the scope of this paper.

Why does the supercompilation produce a more efficient program for the function composition \( \langle F \langle G \rangle \rangle \)? The point is that the embedded call is driven by need, only when the outer call requires more information, and the results produced by the embedded call are immediately consumed by the outer one. So, in the result graph elementary operations of \( F \) and \( G \) are deeply intertwined, and a lot of superfluous operations are thrown away due to the known context.

Why is the supercompilation promising for solving the inverse problem? First, the basic part of the supercompiler, driving, can be used to construct a more reasonable algorithm for searching an argument than the simple SearchArg introduced in the previous section. The idea is that while developing the process graph, terminal nodes, which describe the terminal sets of states, are produced one by one. They are checked, and when the desired one is found, the argument that corresponds to the node is reconstructed by analyzing the walk to this node. If such an argument exists, this algorithm will definitely find it. Second, driving
is in fact symmetrical with respect to direction of computation: from input to output, or vise versa. It can be easily reformulated as development of sets of states from output nodes of a program graph to input ones, moving along arcs in the reverse direction. This gives even more efficient, directed by a goal, process of searching for unknown argument. (A language of so called logic programming, Prolog, is based on similar principles.) Third (the last but not the least), either of interpretive processes of searching for an argument may be metaevaluated (specialized) to produce an efficient one.

### 4.2 Partial evaluation

The DIKU group has found a method for performing specialization, which is surprisingly simple in essence, but gives remarkable results in self-application ((Jones, Sestoft et al. 1985), (Jones, Sestoft et al. 1989), (Jones, Gomard et al. 1993) and a lot of works on improvements of a large community). The method is referred to as "partial evaluation."²

Partial evaluation can be performed if the input to a program is split into several arguments. To run the program they are normally supplied together. However, if only part of them is known, the program can be simplified: those computations that depend only on the known arguments, can be performed in advance. This is just a very naive idea of partial evaluation, and it is not enough for successive metacomputation, but if we consider a program definition to be just one text with arguments being parameters of the whole text, partial evaluation has to do nothing more.

However, the program definition has some internal structure and specific mechanism of its interpretation. Keeping into account this peculiarities allows us to perform a deeper transformation. Partial evaluation exploits two features: First, a program definition consists of separate definitions of subprograms which call each other (one of them, which is considered main, consumes the input of the whole program). Second, reducing specialization of the whole program to specialization of subprograms, several instances of each subprogram are produced. The point is as follows.

During ordinary computation, when some subprogram definition is evaluated and the value of another subprogram called with particular arguments is required, its definition is replicated, the arguments are substituted into the copy, and the test thus obtained is evaluated. Because unlimited number of copies of each subprogram definition can be produced during one run of the program, this mechanism of replication is essential. Going to metacomputation, the mechanisms of ordinary computation are generalized to the case of partially known data. The evaluation of a subprogram turns into evaluation of part of subexpressions, producing not just a value but the specialized definition. The replication of subprograms in order just to evaluate them and throw away, turns into generation of specialized instances.

This is the basic idea of partial evaluation. However, two problems to be solved forces the authors of the method to invent a special control mechanism: finiteness and self-application. The first problem arises because the number of specialized instances may be infinite. To beat the problem the degree of specialization may be decreased (producing in the worst case the source text unchanged). The simplest (and actually used) solution is to prohibit evaluation of those subexpressions which may produces infinite number of values. They are equated to subexpressions depending on unknown values, which are left unchanged. This decision is made in advance either by an automated tool, or by the user, and is supplied to the process of partial evaluation proper by special annotation of a program text.

The second problem, self-application, was solved by performing the control of partial evaluation in advance, by a preliminary stage referred to as binding-time analysis (BTA). The goal is to simplify the process of partial evaluation proper as much as possible by separating out all computation that can be performed in advance. This simplification is very important for self-application. The idea of binding time analysis is as follows. Consider the process of partial evaluation of subprogram: substitute the values of known arguments and evaluate those subexpressions which depend only on them. The processes of substitution and evaluation are rather simple, and they are needed in any case, but the check whether subexpressions depend on just knowns requires additional traversal of the whole text. Fortunately, actual values are not needed for this check, just information which arguments will be known is sufficient. It is just this work that is performed in advance, before actual values are given. Its result is the program text with annotations of subexpressions whether they will be known or unknown. Note that just the same annotation is used to beat infiniteness. So, the whole process of using the specializer looks as follows (as minimum):

1) the user (or special automated tool) marks subexpressions which must be considered unknown in any case (to prevent infiniteness);

2) the user supplies information which arguments will be known at the next stage, and then the BTA distributes this information along the program and marks (annotates) subexpressions whether they will be known or unknown;

3) the user provides values, and the partial evaluation proper is performed.

If the result is not satisfactory, e.g., if the partial evaluation does stop in expected time, the user can return to the step (1), change his annotation and repeat.

Here we see a hierarchy of the basic level (1) and 2 metalevels (2) and (1). It is interesting that their activity is separated in
time: the top metalevel (1) works first and supplies information for the penultimate level (2) which then works and produces information for the basic level (1). The whole process is controlled by the top-most metasystem -- the user.

<Some pictures could be drawn to illustrate this section.>

4.3 Metasystem structure of the methods

Summing-up, each of the two methods has a typical metasystem structure consisting of some basic metaevaluation mechanism and its control.

Basic metaevaluation is generalization of the ordinary evaluation to the case when part of data are unknown, that is, to a set of data instead of a single data. In supercompilation, this is driving, which develops a process tree for a set of input data represented by expression with variables. In partial evaluation, this is the main phase (we referred to it is partial evaluation proper), which creates instances of function definitions and evaluates subexpressions that depends only on known arguments and leaves other subexpressions unchanged. Both, driving and partial evaluation proper, in the case when a set of input data consists of just one value, reduce to ordinary computation.

Control, in supercompilation, is the mechanism which observes the process of driving and makes decisions to loop-back and generalize. In partial evaluation, it is a preliminary analysis (binding-time analysis), which marks the subexpressions that are to be evaluated. The main goal of both controls is to make the process of metacomputation finite. This is a hard task, and for rather intelligent decision-taking it may require control of next metalevels and, in particular, from the user.

The important difference between supercompilation and partial evaluation is that the control in supercompilation is performed on-line, that is, simultaneously with the process of driving, and the control in partial evaluation is off-line, that is, it is performed in advance. Separating off the control process of partial evaluation as a preliminary stage is the main ground of its self-application. This instructive experience could be used in other self-applicable metacomputation. However, performing only such metacomputation that is controlled in advance sets limits to the power of metacomputation: only part of properties could be known before actual values are given. In its turn, being basically on-line, supercompilation can capture dynamic properties of the program, which exhibit themselves only during metacomputation proper.

<A picture could be drawn to illustrate this section.>

5 Manipulating language definitions

In this section we demonstrate series of MSTs that solves the problems of converting different kinds of languages definitions into each other: compilers into interpreters and vise versa. An example of a ultra-MST concludes the section.

It is an inherent property of linguistic modeling that one deals with a variety of different languages for expressing models. We are free to choose whatever formalism is adequate for describing and solving a particular class of problems. Languages are created, used and modified as new problems arise. For example, the researcher working at the scientific frontier can never be sure of finding what he needs in formal languages that has already been applied to previous problems. A discoverer of a new physical theory must have available those formalisms of mathematics that are needed. For example, when Newton found his law of gravitation, he needed to invent the differential calculus and the concept of a second derivative; when Einstein found the general theory of relativity, he needed Riemannian geometry; when Schrödinger found wave mechanics, he needed partial differential equations and eigenfunctions.

There exists no canonical language for linguistic modeling. No language suited for one problem, is necessarily the best for all other problems. Indeed, a myriad of formal languages has been invented. To name just a few examples: formal logic for proving theorems, programming languages for writing algorithms, grammars for describing the syntax of languages.

To say a language has a sense, a semantics, we are to point to an agent that understands the language. He/she/it may be a human being, an animal, or a computer, -- real or imaginable. "Understanding" is demonstrated by performing activities controlled by the language. In the case of a formal language, the controlled activity is established with the greatest possible accuracy.

The relation between a language and its agent is so tight, that we shall not even distinguish them and denote by the same letter, say L. We denote the activity defined by an expression L with some initial data x, performed by an agent L that understands the language L, as follows:

< L x > _ L

Depending on the context, this notation means either the activity proper, the process, or only the result produced after performing it.
Pointing to an agent is just one, the basic, method of defining a language. After several expressive enough, languages exist, we can use them to define new ones. Thus, the world of formal languages extends by its own means. Algorithmic languages forms the foundation of this world.

Definitions of formal languages are linguistic objects as well. We can manipulate them by means of linguistic modeling itself. Let us consider the problem of converting interpretive and compilative definitions of languages into each other as the most illustrative one.

**Defining languages by interpreters and compilers** Two main methods of reducing a language to other languages are used: compilation and interpretation. *Compilation* is translating expressions from one language, say $\mathcal{N}$, to another language, say $\mathcal{L}$:

$$\langle \text{CompNL } n \rangle_{\mathcal{N}} \rightarrow l$$

where $n$ and $l$ are expressions in $\mathcal{N}$ and $\mathcal{L}$ respectively. The compiler is written in the language $\mathcal{M}$. So, the new language $\mathcal{N}$ is reduce to two languages $\mathcal{L}$ and $\mathcal{M}$. The languages $\mathcal{N}$ and $\mathcal{L}$ are referred to as *source* and *target* ones. The language $\mathcal{M}$ in which the definitions of other languages are given is called a *metalanguage*.

*Interpretation* is defining the activity implied by the expression, $n$, in one language, say $\mathcal{N}$, in another language, say $\mathcal{M}$:

$$\langle \text{IntN } n, x \rangle_{\mathcal{N}} \rightarrow y$$

where $x$ is an initial data, $y$ is the same result as produced by an agent understanding the language $\mathcal{N}$:

$$\langle n, x \rangle_{\mathcal{N}} \rightarrow y$$

In the case of interpretation, the new language $\mathcal{N}$ is reduced to one language $\mathcal{M}$.

Thus, we have two definitions of the same language $\mathcal{N}$. Depending on the circumstances, we may want to convert a compiler to an interpreter or vice versa.

### 5.1 Converting compilers to interpreters

Having a compiler of a language $\mathcal{N}$ to a language $\mathcal{L}$, for which an interpreter is provided, we can execute a program $n$ in $\mathcal{N}$ in two stages: first by compiling it to $\mathcal{L}$, then by interpreting the $\mathcal{L}$-program:

$$\langle \text{IntL} \langle \text{CompNL } p \rangle, x \rangle \rightarrow y$$

Thus, the interpreter of $\mathcal{N}$ is defined as follows:

$$\text{def } \langle \text{IntN } p, x \rangle = \langle \text{IntL} \langle \text{CompNL } p \rangle, x \rangle$$

However, the way through another language may sometimes be rather inefficient. In the next section we will discuss an example when the interpretation through compilation is deliberately impossible. So, we may need to optimize the process of interpretation of the new language $\mathcal{N}$.

**First MST** If we metaevaluate the definition above, we will obtain a more efficient interpreter:

$$\times \langle \text{Mc} \_____________ \rangle \rightarrow \text{IntN}$$

$$\langle \text{IntL} \langle \text{CompNL } p \rangle, x \rangle$$

Indeed, $\text{IntN}$ is a program that having got two arguments, $p$ and $x$, executes the program $p$ with the input $x$. (Recall the formal rule described in Section 2: variables appearing at the penultimate level are arguments of the result program.) So, the new interpreter, $\text{IntN}$ is produced by the first MST over the process defined by the composition of $\text{IntL}$ and $\text{CompNL}$.

**Second MST** Every so often we are interested in generating interpreters from a set of compilers, all of which use the same target language $\mathcal{L}$. Then the metacompilation above is to be performed for an arbitrary compiler. To underline that we consider not a particular compiler, $\text{CompNL}$, but an arbitrary one, we replace $\text{CompNL}$ by a variable starting with lower-case letters, $\text{compNL}$; but before, we lift up it to the surface of the MST-formula:

$$\langle \text{Mc} \____________\text{compNL} \________ \rangle \rightarrow \text{IntL} \langle \times \ p \rangle, x \rangle$$
This MST-formula defines the process of converting a compiler, \( \text{compNL} \), to the interpreter. This process can be optimized by metacomputation:

\[
\begin{align*}
\text{Mc} & \quad \text{compNL} \\
\text{Mc} & \quad \text{IntL} \ < \ * \ p, \ x >
\end{align*}
\]

The result, \( \text{IntNGen} \), is an algorithm, which applied to a compiler from a language \( N \) to the fixed target language \( L \), produces the interpreter of the source language \( N \):

\[
\text{IntNGen} \ \text{CompNL} \Rightarrow \ \text{IntN}
\]

**Third MST** We may wish to vary the interpreter of the target language, \( \text{IntL} \), in the MST-formula above (**) . To express our wish, we lift up \( \text{IntL} \) to the top level of MST-formula and replace the capitalized identifier that denotes the constant, \( \text{IntL} \), by a lowercase variable, \( \text{intL} \):

\[
\begin{align*}
\text{Mc} & \quad \text{intL} \\
\text{Mc} & \quad \text{compNL} \\
\text{Mc} & \quad \ast \ < \ * \ p, \ x >
\end{align*}
\]

This MST-formula defines the process that given an interpreter, \( \text{intL} \), of the target language, \( L \), produces a generator of interpreters of languages defined by their compilers to \( L \). Now, we want to make this process more efficient and metaevaluate the formula above:

\[
\begin{align*}
\text{Mc} & \quad \text{intL} \\
\text{Mc} & \quad \text{compNL} \\
\text{Mc} & \quad \ast \ < \ * \ p, \ x >
\end{align*}
\]

The result, \( \text{IntNGenGen} \), is an algorithm that given an interpreter, \( \text{intL} \), of a target language \( L \), produces a generator of interpreters from compilers that uses the language \( L \) as a target one:

\[
\text{IntNGenGen} \ \text{IntL} \Rightarrow \ \text{IntNGen}
\]

Of course, the quality of the produced interpreters, interpreter generators and generator of interpreter generators depends upon the state of art of the metacomputation methods implemented in \( \text{Mc} \). In the present moment real computer experiments are not performed yet, though very promising results have been achieved already.

In Section 6.1 we shall return to this topic and demonstrate the need of converting compilative definitions to interpretive ones in mathematics. Generally, this is needed when a language extends gradually; in this case compilative definitions are usually simpler than interpretive ones. However, when a rather different new language is defined, it is much easier to write out an interpreter than a compiler. In such a case, the opposite problem arises: the conversion of an interpreter to a compiler. While mathematics mainly deal with the former case, in programming the latter is met more often. Let us turn to it. It is of special interest because it is put into the computer already.

### 5.1 Converting interpreters to compilers

Having an interpreter of a language \( N \) written in a language \( M \), we can execute a program \( \text{P}_N \) in \( N \) using the \( M \)-machine:

\[
\text{IntN} \ \text{P}_N \ \text{x} \Rightarrow \ y
\]

where \( \text{x} \) is an argument, \( y \) is the result.

**First MST** This formula gives the definition of the process for the machine \( M \), that is, in fact, in the language \( M \). To derive an explicit efficient representation in \( M \), we metaevaluate the formula:

\[
\begin{align*}
\text{Mc} & \quad \text{P}_N \\
\text{Mc} & \quad \text{IntN} \ \text{P}_N \ \text{x}
\end{align*}
\]

The result, \( \text{P}_N \), is a program in the language \( M \) equivalent to \( \text{P}_N \):
Second MST  The compilation defined by the formula \((*)\) can be performed for any program in \(\mathbb{N}\). Let us explicate this, lifting up \(\mathbb{P}_\mathbb{N}\) and replacing it by a lower-case variable, \(p\):

\[
\langle \text{IntN} \; p \; x \rangle \rightarrow \langle \mathbb{P}_\mathbb{N} \; x \rangle
\]

This is the definition of the process of compilation from \(\mathbb{N}\) to \(\mathbb{M}\) with an argument \(p\). We can metaevaluate it to construct an efficient compiler:

\[
(**) \quad \langle \text{MC} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \Rightarrow \text{CompNM} \\
\langle \text{MC} \; \underline{\text{p}} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \\
\langle \text{IntN} \; x \; x \rangle
\]

The result, \(\text{CompNM}\), is an algorithm, which applied to a program \(\mathbb{P}_\mathbb{N}\) in the language \(\mathbb{N}\) returns an equivalent program in \(\mathbb{M}\), that is, it is actually a compiler from the language \(\mathbb{N}\) to the language \(\mathbb{M}\):

\[
\langle \text{CompNM} \; \mathbb{P}_\mathbb{N} \rangle \rightarrow \mathbb{P}_\mathbb{N}
\]

Third MST  The compiler generation defined by the formula \((***)\) can be performed for any interpreter. Let us explicate this, lifting up \(\text{IntN}\) and replacing it by a lower-case variable, \(\text{int}\):

\[
\langle \text{MC} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \\
\langle \text{MC} \; \underline{\text{p}} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \\
\langle \text{IntN} \; \underline{x} \; \underline{x} \rangle
\]

This is the definition of the process of generating the compiler from an interpreter. Let us metaevaluate it to construct an efficient compiler generator:

\[
(***') \quad \langle \text{MC} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \Rightarrow \text{CoGen} \\
\langle \text{MC} \; \underline{\text{p}} \; \underline{\text{int}} \; \underline{\text{int}} \rangle \\
\langle \text{IntN} \; \underline{x} \; \underline{x} \rangle
\]

The result, \(\text{CoGen}\), is an algorithm, which applied to an interpreter \(\text{IntN}\) of some language \(\mathbb{N}\) returns an a compiler from the language \(\mathbb{N}\) to the metalanguage \(\mathbb{M}\):

\[
\langle \text{CoGen} \; \text{IntN} \rangle \Rightarrow \text{CompNM}
\]

So, the \(\text{CoGen}\) is a compiler generator in fact.

As usual in metacomputation, the quality of the produced compilers and compiler generator depends upon the state of art of the metacomputation methods implemented in \(\text{MC}\). The first compiler generator was produced in 1985 by the specializer based on partial evaluation, the pure functional subset of the language \text{Lisp} using in role of metalanguage \(\mathbb{M}\) (Jones, Sestoft et al. 1985). Thereafter, due to a lot of improvements of the method, the quality of result programs was increased and the compiler generators was implemented for some other programming languages. Of course, a lot of special problems of compiler construction are not solved by these MST-formulas, e.g. the problem of code generation. However, it is a considerable step forward in the automation of program and compiler construction, and in manipulating linguistic models of interpretive nature.

Ultra-MST  The MSTs in this section proceed by a common schema: Starting with some basic formula, e.g. \(<\text{IntN} \; \langle \text{CompN} \; p \rangle \; x \rangle \) or \(<\text{IntN} \; \mathbb{P}_\mathbb{N} \; x \rangle \), one performs as follows:

a) consider some part(s) of the formula varying: lift up it to the surface and replace by a variable(s), which we denote by lower case. (For our basic formulas this step was performed in advance: \(p\) and \(\chi\) for the former, and \(\chi\) for the latter, were picked out from the very beginning);

b) perform metacomputation of the resulting formula: denote this by writing \(\langle \text{MC} \; \underline{x} \rangle \) above it.

Then we may return to the step (a) with the MST-formula obtained.

Having formulated a schema that controls a series of MSTs, we have performed a ultra-metasytem transition (Turc...
1977). Now the schema can be applied either indefinitely, or until it becomes inapplicable. What will happen in our case? What MSTs can be performed more?

First three MSTs exploit all parts of basic formulas to become variables. In principle, we could choose another order of replacing them by variables. We have not considered these cases because they have minor significance. However, can the forth, the fifth and so on, MSTs be performed? Yes, after each MSTs, one more Mc appears in the formula, and we can abstract some of its occurrences. Let us consider this formally.

**Forth MST:**

\[
\langle \text{Mc} \rangle \quad \text{CoGenGen}'
\]

**Fifth MST:**

\[
\langle \text{Mc} \rangle \quad \text{CoGenGen}''
\]

**Sixth MST:**

\[
\langle \text{Mc} \rangle \quad \text{CoGenGen}'''
\]

This seems to be a useless game. Nevertheless, these MST-formulas can be used in the following practical situation. When a new metacomputation method is implemented by a new Mc', the compiler generator is to be recomputed using the formula (***):

\[
(***') \quad \langle \text{Mc}' \rangle \quad \text{CoGen}''
\]

However, it may be generated using CoGenGen'' produced by the sixth MST as well:

\[
\langle \text{CoGenGen}'' \rangle \quad \text{Mc'} \quad \text{CoGen}' \quad \text{CoGen}''
\]

In principle, applying CoGen'' to Mc' 3 times may be quicker than evaluating compiler generator from scratch by (***)'. However, generating CoGenGen'' requires computer resources as well, and this may be advantageous if evaluating CoGen'' is performed many times while developing Mc'.

Applying CoGenGen' and CoGenGen'' to Mc' has sense as well:

\[
\langle \text{CoGenGen}' \rangle \quad \text{Mc'} \quad \text{CoGen}'
\]

These CoGen's are "partially improved" as compared with the old CoGen. This is immediately seen if we look at the formulas defining them from scratch:

\[
\langle \text{Mc} \rangle \quad \text{CoGen}'
\]

\[
\langle \text{Mc} \rangle \quad \text{CoGen}''
\]
The former, CoGen', generates a compiler that produces programs of better quality (provided, of course, CoGen' is better than CoGen), but the compiler itself and CoGen' are of previous quality. The latter, CoGen'', generates a compiler which works with the quality provided by CoGen', but has the quality of the old CoGen on its own.

It seems that the forth, fifth, sixth MSTs, and maybe the subsequent ones as well, produce different, though similar, metasystems. However, they are not, and we can escape evaluating CoGen''' and use CoGen obtained by the third MST instead. Compiler generator, CoGen, can be applied not to just ordinary interpreters of algorithmic languages, but to any linguistic processor of interpretive nature, to the metacomputation processor, CoGen, in particular. To express this possibility more clear, let us redefine CoGen as interpreter (having named it Spec), which has two arguments, a program, p, and a data, x:

\[
\text{def Spec } p, x = \langle \text{CoGen' } p, x \rangle
\]

The so defined, Spec, specializes the program of two arguments, p, with respect to the first argument, x. The CoGen applied to the Spec produces the "partially improved" compiler generator, CoGen', which applied to the Spec once more produces the "more improved" CoGen'', which applied to the Spec for the third time produces the renewed compiler generator, CoGen'''.

\[
\begin{align*}
&\langle \text{CoGen Spec} \rangle \\
&\rightarrow \langle \text{CoGen } \rangle
\end{align*}
\]

\[
\begin{align*}
&\rightarrow \langle \text{CoGen Spec} \rangle \\
&\rightarrow \langle \text{CoGen' } \rangle
\end{align*}
\]

\[
\begin{align*}
&\rightarrow \langle \text{CoGen Spec} \rangle \\
&\rightarrow \langle \text{CoGen''} \rangle
\end{align*}
\]

Summing-up, the process of generating the new compiler generator, CoGen''', from an existing one, CoGen, and a new metaevaluator in form of specializer Spec is expressed as follows (Klimov and Romanenko 1987):

\[
\langle \langle \langle \text{CoGen Spec Spec Spec} \rangle \rangle \rangle = \text{CoGen'''}
\]

We have considered an example of a ultra-MST that comes to a fixed point at the forth step; that is, all MSTs starting with the forth one have the identical control subsystems: they produces a "partially improved" compiler generator by applying the previous one to a new metaevaluator. However, only one, two and three performances of this MST has practical sense.

### 6 Applications to mathematics

This section contains the material of two kinds. First two subsections are devoted to applications of general MST-schemata in linguistic modeling to mathematics: in Section 6.1 we consider the problem of manipulating hierarchies of mathematical definition as the problem of converting compilative definitions into interpretive ones, and in Section 6.2 we demonstrate how some class of mathematical theorems may be proven by metacomputation of composition of functions one of which is metaevaluator itself. In the last subsection we review the ideas behind Turchin's Cybernetic Foundation of Mathematics, paying special attention to its metasystem structure.

#### 6.1 Compressing hierarchies of mathematical definitions

The most illustrative example of the need of converting a compiler to the interpreter is presented by mathematics. Each definition in a mathematical textbook is a compilative one. It says that the formal meaning of the notion is that each its occurrence must be replaced by the expression provided by the definition. Of course, mathematicians do not perform such substitutions in their minds while working, they use some informal images, some mental interpretation. But formally the mathematical theories are very high
hierarchies of compilative definitions.

In 1970 \[ ? \], V.Turchin and S.Romanenko has performed an instructive experiment based on the famous treatise by Nicholas Bourbaki "Elements of mathematics" [What is the title in English? Maybe, "Fundamentals..."?]. The first chapter of the first book, "The set theory", defines the basic language of the whole mathematics described in the treatise and the notion of a correct proof. The later is provided by an algorithm that check whether a given text, \( t \), in the basic language is a correct proof (we call this algorithm BasicProof, though it was not named by Bourbaki):

\[
<\text{BasicProof } t> = \text{True or False}
\]

The text of the treatise is a sequence of definitions and proofs expressed in terms of defined notions and proof patterns, which are abbreviations of phrases in the basic language. The sequence of definitions, \( D \), is a compiler description, which supplied to an algorithm that expands definitions, say DefComp, translates a text, \( t \), to the basic language:

\[
<\text{DefComp } D. t> = t'
\]

A text \( t \) is considered to be a correct proof, if the result of its compilation to the basic language is a correct proof:

\[
<\text{BasicProof } <\text{DefComp } D. t>> = \text{True or False}
\]

So, we have the interpreter, BasicProof, which assigns a meaning to phrases in the basic language, and the compiler, DefComp, parameterized by \( D \), which defines the meaning of an extended language in terms of the basic one. However, the author and the reader of the treatise do not actually perform compilation. They have some informal mental interpretation of the mathematical text, that is, of \( D \) and \( t \), which make them to believe that if DefComp and BasicProof are evaluated indeed, the result will be True.

Then the following idea comes to one's mind: Why not to evaluate this by the computer? V.Turchin and S.Romanenko made an attempt to do this. And they failed! What was the problem? They have taken a mere first texts from the first chapter and tried to expand them into the basic language, but the computer memory was exhausted very quickly. After analyzing the trace of computation, they saw that any realistic memory will be exhausted if not for the texts from the first chapter then from the second, because results grow exponentially of the height of the hierarchy of defined notions.

This showed that nobody actually checked and even cannot check the treatise of Bourbaki directly. Mathematicians interpret the text by their minds without compiling it to the basic language, but the rules of this interpretation are not explicated in the treatise. This is not occasional: when a language is extended gradually adding notions one by one, it is much simpler to give a rule for compilation to the known notions than to define formal rules for interpreting a text in the extended language.

So, the problem arises to derive an interpreter of the higher language, Proof, from the interpreter of the basic language, BasicProof, and the compiler, DefComp, parameterized by definitions, \( D \).

**The first MST** We have the interpreter fully specified:

\[
\text{def } \langle \text{Proof } t \rangle = <\text{BasicProof } \langle \text{Comp } D. t \rangle>
\]

but to become executable well, its definition is to be metaevaluated:

\[
(*) \langle \text{Mc } \ldots \rangle = \text{Proof}
\]

\[
\langle \text{BasicProof } <\text{DefComp } D. t>>
\]

**The second MST** Now let us repeat the way of the previous section and vary the compiler definition. This is just what the Bourbaki's formalization of mathematics requires: After each mathematical definition, the compilation process parameterized by a set of definitions, \( d \), changes, and a new proof-checker should be constructed (the letter \( D \) is lowered to denote a variable):

\[
<\text{Mc } \ldots d \ldots >
\]

\[
<\text{BasicProof } <\text{DefComp } *, t>>
\]

After the notion of the basic proof, BasicProof, and the algorithm of compiling definitions, DefComp, are defined in the first chapter of the first book by Bourbaki, the process above can be metaevaluated:

\[
(**) \langle \text{Mc } \ldots d \ldots > = \text{ProofGen}
\]

\[
\langle \text{Mc } \ldots d \ldots >
\]

\[
<\text{BasicProof } <\text{DefComp } *, t>>
\]
The result, ProofGen, is a generator of interpreters of mathematical texts that use a fixed set of notions; that is, an algorithm, which given definitions, D, produces the algorithm, ProofD, which checks whether a given mathematical text using the notions defined in D is a correct proof:

\[ \text{<ProofGen D>} \Rightarrow \text{ProofD} \]
\[ \text{<ProofD t>} \Rightarrow \text{True or False} \]

**Varying the basic language** Though the basic language does not change in the Bourbaki's treatise, after each definition, \( D_{n+1} \), is given, the extended language may be considered a new basic one. Therefore, a new proof-checker, \( \text{Proof}_{n+1} \), can be generated from the previous one, \( \text{Proof}_n \):

\[ (*) \quad \text{<Mc \hspace{1cm} DEF\hspace{1cm} DefComp D_{n+1} t> } \Rightarrow \text{Proof}_{n+1} \]
\[ \text{<Proof\hspace{1cm}n\hspace{1cm}DefComp D_{n+1} t>} \]

It would appear reasonable that this process of incrementally generating a proof-checker is faster than evaluating \( \text{Proof}_n \) from the basic language interpreter:

\[ \text{<Mc \hspace{1cm} DEF\hspace{1cm} BasicProof (DefComp D_{n+1} D_{n+1} t>) } \Rightarrow \text{Proof}_{n+1} \]

and it may be either faster, or slower than using the proof-checker generator, \( \text{ProofGen} \), depending on the complexity of the list of definitions \( D_1 \ldots D_n \hspace{1cm} D_{n+1} \):

\[ \text{<ProofGen D_{n+1} D_{n+1} t> } \Rightarrow \text{Proof}_{n+1} \]

By analogy with the formula (**) , the generator of incremental proof-checkers can be constructed, if a mathematical theory defined by \( \text{Proof}_n \) is to be extended in several ways:

\[ (**) \quad \text{<Mc \hspace{1cm} DEF\hspace{1cm} Mc d> } \Rightarrow \text{ProofGen}_n \]
\[ \text{<Mc \hspace{1cm} DEF\hspace{1cm} Proof\hspace{1cm}n \hspace{1cm} DefComp * t>} \]

The result algorithm, \( \text{ProofGen}_n \), applied to definitions, \( D_{n+1} \), produces the extended proof-checker \( \text{Proof}_{n+1} \):

\[ \text{<ProofGen\hspace{1cm}n\hspace{1cm}D_{n+1} \hspace{1cm} t> } \Rightarrow \text{Proof}_{n+1} \]
\[ \text{<Proof\hspace{1cm}n\hspace{1cm}t> } \Rightarrow \text{True or False} \]

**The third MST** We can consider the process (**) for an arbitrary proof-checker \( \text{Proof}_n \) (we lift it up and replace by a variable, \( p \)):

\[ \text{<Mc \hspace{1cm} DEF\hspace{1cm} Mc p \hspace{1cm} d} \]
\[ \text{<Mc \hspace{1cm} DEF\hspace{1cm} Proof\hspace{1cm}n \hspace{1cm} DefComp * t>} \]

After metaevaluating this process:

\[ (*** \quad \text{<Mc \hspace{1cm} DEF\hspace{1cm} Mc p \hspace{1cm} d} \Rightarrow \text{ProofGenGen} \]
\[ \text{<Mc \hspace{1cm} DEF\hspace{1cm} Proof\hspace{1cm}n \hspace{1cm} DefComp * t>} \]

we obtain the generator of incremental generators of proof-checkers, which applied to a proof-checker, \( \text{Proof}_n \), produces the same incremental generator as that by the formula (**):
Another second MST. It may seem that the series of MSTs appears to be deterministic. However, they were chosen due to just our will. Considering the process of incrementally generating a proof-checker (*), we can vary not just one \( \text{D}_{n+1} \) but both \( \text{P}_n \) and \( \text{D}_{n+1} \) (they are replaced by variables \( p \) and \( d \) respectively):

\[
\text{proof}_n \quad \text{and then, performing the second MST, produce the incremental proof-checker generator:}
\]

\[
\text{ProofIncGen} \quad \text{Proof}_n \quad \text{D}_{n+1} \quad \text{Proof}_{n+1}
\]

**Conclusion.** All of the considered MST-formulas are based on the following metacomputation operations: program composition, specialization and MST (self-application). It is clear that if the metacomputation methods are powerful enough to compress mathematical hierarchies of definitions, then they will be useful for solving a lot of practical problems. The main severity here is successive metacomputation of program composition.

We saw also that MSTs can form not only linear series, but that several MSTs can be performed if a subject activity allows varying of several parts. In the case of definition hierarchies, a series of languages is produced by gradual extensions. The additional dimensions for MSTs appeared from this feature. In fact, even more MST formulas can be written out to formalize another problems, which are beyond the scope of this paper.

6.2 Theorem proving by metacomputation

In this subsection we show how some class of mathematical theorems may be proven by metacomputation (Turchin 1980). The development of the supercompilation methods is especially governed by this application.

The classic axiomatic method of representing mathematical theories as linguistic machines was discussed in the previous section. However, it is not the best for manipulating by metacomputation. Instead of writing out axioms which state properties of some objects and functions, and then defining the algorithm checking the proof, it is more appropriate to represent objects as linguistic expressions, and functions and predicates as programs. Then metacomputation can be applied to them directly.

For example, to define the notion of a natural number we may represent numbers in 'unary system', number \( n \) being a string consisting of a symbol 0 followed by \( n \) apostrophes: \( '0' \); and instead of the Peano axioms for formal arithmetic, we should write out the definitions of addition and equality (we hope the notation is self-evident):

```python
def <Add x, 0> = x
<Add x, y'> = <Add x, y>'

def <Eq 0, 0> = True
<Eq x', y'> = <Eq x, y>
<Eq x, y> = False
```

Then, for example, the commutativity of addition:

\( x + y = y + x \)

is to be expressed as the statement that the predicate \( P \) defined by the following algorithm returns \( \text{True} \) for any argument:

```python
def <P x, y> = <Eq <Add x, y>, <Add y, x>>
```

This is the idea of the constructive approach as opposed to the formal axiomatic one.

Proving by metacomputation. In such a way, any theorem can be expressed, which has the form:

"For all \( x, y, ..., z \), \( P(x, y, ..., z) \) is true."
where \( P \) is a computable predicate. To try to prove the theorem, we metaevaluate the predicate \( P \):

\[
\text{def } \langle P \ x, y, \ldots, z \rangle \Rightarrow \text{True}
\]

and if the text of the result algorithm \( P' \) has a special form, from which it is clear that it is always True, we conclude that the theorem holds. The simplest such a form is evidently as follows:

\[
\text{def } \langle P \ x, y, \ldots, z \rangle \Rightarrow \text{True}
\]

If the result algorithm, \( P' \), is not of the expected form, this does not mean that the theorem is wrong, but this merely means that it has not been proven yet, that is, the power of the metacomputation methods realized in \( M_C \) is insufficient to prove it.

**Metasystem structure of both representations** The axiomatic representation of mathematical knowledge consists of two parts: axioms (or axiom schemata) and a deductive system. They are organized in a metasystem with the former being a subject subsystem, the latter a control subsystem. Traditionally, mathematicians try to express the majority of knowledge in the subject subsystem, by the axioms. The deductive system is minimized and usually consists of just one rule of deduction, Modus Ponens:

\[
P \quad P \text{ implies } Q \quad \text{two statements which are proved already}
\]

\[
\text{-----}
\]

\[
Q \quad \text{the statement which becomes proved}
\]

In constructive approach, instead of axioms, the subject subsystem consists of algorithms for functions and predicates. Instead of deduction rule(s), the control is the metacomputation processor. So, the 'division of labor' is inverse: although the definitions of basic functions are usually similar to axioms, the control is abundant. Moreover, it is not fixed, it evolves. Each \( M_C \) can't prove all theorems: this is algorithmically unsolvable; but every new \( M_C \) will do more and more.

The control by metacomputation includes some mathematical knowledge that is traditionally represented by axioms. The principle of induction is such one.

**Principle of induction** Traditionally the principle of induction is formulated in form of an axiom. Although the idea of induction is the same for all theories, it is formulated for each new theory from scratch. For example, for the arithmetic it looks as follows:

For all predicates \( P \),

- if \( P(0) \) and for all numbers \( n \), \( P(n) \) implies \( P(n+1) \)
- then for all numbers \( n \), \( P(n) \).

This definition uses particular constructors of the data domain, \( 0 \) and \( n+1 \). In another theory with a data domain built from other constructors, the antecedent takes another form.

This looks strange because the idea of induction is general: if the data domain is built by repeatedly applying some constructors (like \( n+1 \)) starting from atomic data (like \( 0 \)), and if a property holds for all atomic data and is preserved by application of all constructors, then it holds for all data. However, it can't be formalized in form of a universal axiom once for all theories. The principle of induction is a meta-principle, rather than a special axiom. This is indeed the case in the constructive approach, where the induction is formalized as follows.

As we saw, proving a theorem by metacomputation consists of two steps:

1) metacomputation of the predicate which is asserted to be always true;

2) recognition that the result has the form, for which being true is evident.

The power of proving can be increased by improving the methods either of metacomputation, or recognition. Although, in principle, they are the same thing, some improvements are more natural to be formalized in metacomputation, others in recognition. The induction is the latter case.

So, the **principle of induction** is the following recognition rule: If in the graph of the result program all terminal nodes are True, than the program always returns True. Lets us name it \( \text{Ind} \):

\[
\langle \text{Ind } 'a \text{ graph with terminal nodes being True'} \rangle \Rightarrow \text{True}
\]

Otherwise, it returns the answer \( \text{NotProved} \).
The definition of the whole process of proving with induction is then as follows:

\[(\star) \, \langle \text{Ind} \, \langle \text{Mc} \, \ldots \rangle \rangle \rightarrow \langle \text{P} \, x, y, \ldots, z \rangle\]

**Example** For a particular case of commutativity of addition with 1:

\[x + 1 = 1 + x\]

even a rather simple supercompiler will transform the definition of the corresponding predicate:

\[\text{def} \, \langle \text{P} \, x \rangle = \langle \text{Eq} \, \langle \text{Add} \, x, 1 \rangle, \langle \text{Add} \, 1, x \rangle \rangle\]

into the following form:

\[\text{def} \, \langle \text{P} \, 0 \rangle = \langle \text{True} \rangle \]
\[\langle \text{P} \, x' \rangle = \langle \text{P} \, x \rangle\]

The latter is a program with one terminal node being \text{True}. Thus, the theorem is proven.

**Second MST** As in the previous sections, we can metaevaluate the MST-formula above \((\star)\) to make the process of proving more efficient. Before, we are to choose a part to be varied. It will be P. We express this by lifting up it, replacing by a lower case variable p:

\[\langle \text{Mc} \, \ldots \rangle = \langle \text{Ind} \, \langle \text{Mc} \, \ldots \rangle \rangle \rightarrow \langle \text{IndProver} \rangle \]
\[\langle \text{P} \, x, y, \ldots, z \rangle \]

The result of metacomputation, \text{IndProver}, is a prover by induction. It accepts the definition of a predicate p, and try to prove that it is true for all arguments.

The later MST-formula is metacomputation of composition of the predicate \text{Ind} and the \text{Mc} itself. The expected benefit of \text{IndProver} over evaluating the formula \((\star)\) directly is that the embedded process of metacomputation p is immediately canceled when it becomes clear that one of the terminal nodes is not \text{True}.

**Conclusion** We saw that metacomputation can be used as metasystem which (1) defines what means to be a proven statement, (2) actually proves statements on the computer. Of course, the class of provable statements is always smaller than the class of theorems of any theory that is powerful enough. This class grows while methods of metacomputation evolve. This constructive approach to automation of proving theorems by computer seems to be more direct and more promising than traditional approach of artificial intelligence research based on the search of proofs in deductive systems.

### 6.3 The Cybernetic Foundation of Mathematics

In this section the Cybernetic Foundation of Mathematics by V.Turchin (Turchin 1987) is reviewed. Special attention is paid to the metasystem structure of this theory.

Mathematics is actually a science investigating linguistic models. However, the modern mathematics (to be precise, the axiomatic-logic approach which is prevailing today), is mainly interested in finding properties of and relations between formal objects. It is interested in laws about 'ideal entities' which exist somehow in abstracto (such as Cantor's uncountable sets). Modern mathematics remains Platonic in essence (according to which all things are reflections and manifestations of ideas which exists in an ultimate and eternal world).

Computer science is different. Symbols, linguistic processes and linguistic machines constitute the primary reality. Though during last two decades its foundation in Platonic style has been developing as well (e.g. set-theoretical denotational semantics), it is hard to be considered natural. On the contrary, we believe that with the advent of the computer the constructive treatment of problems will become increasingly important.

**Formalistic and constructive approaches** In the 20th century, two main approaches to the foundation of mathematics were proposed:

(i) The first method, the formalistic approach, was introduced by David Hilbert (1862–1943) in the twenties. It has become the foundation of modern mathematics. The idea is that any mathematical proof is a text in some language and that the correctness of a proof can be defined by a precise procedure checking the proof. That is, correctness is defined by an
independent algorithm behaving in exactly the same way each time it is used. Thus, the notion of a correct proof became objective and independent of the human. What the proof and the theories are all about is not essential. Modern mathematics developed according to Hilbert's solution of the mathematical crisis that happens at the beginning of the century. In fact, modern mathematics lacks the objective notion of an object, though mathematicians use some mental images for objects. A rather abstract notion of a set is regarded as sufficient and precise enough to serve as a means to represent objects of particular theories.

(ii) Another approach was developed by Brouwer, Markov and others as a result of a philosophical enterprise arising from certain views concerning the relation between mathematics, the physical world, and mental reality. Although the views of the pioneers differed in various respects, there is sufficient common ground to support a task that can be described roughly as the constructive redevelopment of mathematics. Their idea was to reduce all objects in mathematics to algorithms, that is, only algorithmically defined objects are allowed. This concerns objects and proofs of propositions about objects as well; e.g. if we have propositions that some object exists for some class of situations we must be able to suggest an algorithm which, given a description of this situation, builds the object. In the intuitionist theory of mathematics, founded by Luitzen Brouwer (1881--1966), the concept of number is based on man's intuitive sense of the concept of time. Brouwer, Bishop, and their fellows thought that constructive mathematics should replace ordinary, set-theoretic mathematics. They tried to find objects and method for proofing which are more intuitive. Ultimately they came close to the ideas which were proposed by constructivists such as Andrei Markov (1903--1979).

The constructive approaches of Brouwer, Markov, and their fellows turned out to be relatively limited because they could not represent all set-theoretic concepts. There are objects mathematicians need and which can't be represented by algorithms. For example, one of the most important questions about algorithms cannot be answered algorithmically: whether a computation process will stop in a finite time or not. On the other hand, Hilbert's approach actually covers the needs of present-day mathematics.

Platonic mathematics and computer science Three main questions are to be answered in the foundation of any mathematical theory (and computer science respectively):

i) What are (mathematical) objects?

ii) What is the language of a (mathematical) theory?

iii) How is the notion of a correct proof expressed in a (mathematical) language?

Hilbert's solution answers only the last two questions. Modern mathematicians have no universal notion of what their objects are. That is, mathematics remains a discipline without objects, although, of course, intuitive images exists in the minds of mathematicians. This is why modern mathematics remains Platonic in essence. Hilbert's solution can be regarded as a partial, temporary one, which is suitable to continue working, but the fundamental question remains 'What are mathematical objects?'.

This situation has always been considered unsatisfactorily by constructivists such as Brouwer and Markov. But the potential richness of constructivist studies was not fully realized until general-purpose computing machines became available. Computers were really necessary before we could start to learn more about the general properties of linguistic models; human beings are not precise enough nor fast enough to carry out any but the simplest procedures. Indeed, the impact of computer science is an increased emphasis on constructions in all branches of mathematics. The question remains open, whether the view of modern mathematics may be helpful in building the foundation of computer science and to take advance linguistic modeling as a scientific method where tangible objects, such as symbols and machines, are the primary matter.

Cybernetic Foundation of Mathematics In the Cybernetic Foundation of Mathematics Turchin has shown that the full set theory can be built constructively without referencing a Platonic world, only by using linguistic material (Turchin 1987). This shows how to abandon the limitations of pure constructivism by metasystem transition over algorithms. Turchin actually converged two directions: Hilbert's with the notion of language which provides the material in which theories were expressed - set theory as the most central example - and Brouwer's and Markov's constructivist direction. The viewpoint of classic constructivists concerns just the first level of Turchin's MST tower: the ground level objects which are algorithms. In the Turchin's Cybernetic Foundation the processes are organized in metasystem tower, and the algorithms are extended to processes which may, in addition, use algorithmically unsolvable predicates over the processes of lower levels. They are referred to as metamechanical processes (as opposed to mechanical processes being just ordinary algorithms).

In the remaining part of this section we will discuss two essential concepts of the Cybernetic Foundation of Mathematics: (i) the concept of truth and (ii) the concepts of a real and the ideal user.

The concept of truth In the previous section about theorem proving we have introduced predicates which are expressed algorithmically and returned either True or False. However, in the Cybernetic Foundation of Mathematics no predicates about objects are used. To underline the difference the statements are referred to as predictions, and they do actually predict facts about behavior of linguistic machines. There are two kinds of predictions to which all others are reduced: the prediction that a process will stops, and that it never stops. This is indeed the basic and elementary truth about machines. All other facts, such as predicates returning either True or False, are reduced to such predictions. However, the problem is how such predictions
about computation processes are decided to be true or false.

(i) If it is predicted that "a process will stop", and after some time the process actually stops, then the original prediction is proven. As long as the process does not stop, nothing can be said about the prediction. If the prediction is false, no falsification can be obtained in a finite time just by observing it. The prediction "a process will stop" has an objective meaning which can be proven, but it cannot be falsified directly.

(ii) If it is predicted that "a process never stops", only the answer False can be obtained in a finite time (when the process stops, it falsifies the prediction). This prediction can be falsified, but it cannot be approved directly.

The incompleteness of the basic means to evaluate predictions is solved on the next metasystem level. What is actually needed is to explain how a prediction can be proven or refuted in a finite time during the process of using the linguistic model. The notion of time (of the user) is a crucial difference with the Platonic viewpoint.

**The concept of a real and the ideal user** In the later of the above cases (non-termination), and in order to continue execution of the process that asks whether the prediction is True or False, the user must supply additional knowledge to force the prediction to be evaluated in finite time.

This can be achieved by two parallel processes: one executing the process under question, the other searching through the facts supplied by the user. At each particular moment the store is finite, the user can inspect the facts searched in the store and may add new facts. That is, the size of the store is unbound due to the parallel activity of the user.

Another problem is, how to ensure that the user does not lie? The question is answered by distinguishing between a real user and an ideal user. Assume that a real user supplied an incorrect statement of the form "process \( P \) does not stop" but process \( P \) has actually stopped, then the error is noticed in a finite time. What does the real user do in such a situation? He says: "My hypotheses was wrong. Let's return, restore the state of the machine before I issued the prediction and take the opposite hypothesis" (which, by the way, is already known to be true). It is important to note that each error of the real user is found in a finite time. Therefore, each erroneous branch in the path of a real user is finite. That is, by trial and error, the real user finds the right set of facts about processes.

If all erroneous branches are cut off from the path of the real user, we'll get a path for a user who always issues non-falsifiable predictions. This is the ideal user. Thus, the notion of truth involves the notion of the user.

**Do algorithms stop uniformly?** There is another question: Does one believe that processes stop uniformly for all users? It is easy to believe in this statement for algorithms, because it is the basic belief, on which all mathematicians agree and on which the Hilbert's approach relied as well. Mathematicians agree on this although it remains an unprovable statement.

However, for a metamechanical process this is a non-trivial belief. Predictions about metamechanical processes are not objective in general; the famous paradoxes, such as the Paradox of a Lier can be formalized as a process of such kind. Though we can not escape relying on belief, things to be believed in must be listed explicitly in order to give everyone an opportunity to decide himself. Turchin has constructed a class of metamechanical processes that are believable to be 'objectively interpretable' and sufficient to interpret the set theory.

The Cybernetics Foundation introduces the user as the second central concept, and use it in constructs which are believable to be objective. The notion of user might appear surprising, but we should consider that the transition from mechanistic view of the physical world to the theory of relativity was marked by incorporating the notion of observer. One of the essential results of the 20th century physics was that the observer can not be separated from the world of observed matters. That is, in the last analysis there is no objective notion for physical processes. In the same way, the user of the linguistic world, cannot be removed from the matter he uses and observes.

**Ultra-metasystem transition** Objects in the Cybernetic Foundation of Mathematics are organized in a metasystem hierarchy. Objects at each level are algorithms which can incorporate statements about algorithms at the previous level. Objects at the first level are plain algorithms. Whether a statement about a process is true or not, can not in general be decided by a metamechanical process of the same level, hence in the metaactivity additional knowledge may be needed which the user of the system supplies. A step from a level to the next one is an MST, which involves the user.

So, the Cybernetic Foundation as a whole has a structure of an ultra-metasystem. As opposed to classic constructive mathematics, as well as to axiomatic theories, it is open and is not based on finite number of facts. "This is why it is free of Gödel limitations, what other limitations it has we don't see yet." (Turchin 1980, last page)

**Cybernetic Foundation and the evolution of science** The limitation of previous approaches of Brouwer and Markov is that they assume that only facts which are clear to be true can be used. But Turchin's introduction of the user involves trial and error (and back-tracking), and allows us to use facts that are proven in a finite number of steps as well as statements which cannot be proven but that can be falsified in a finite number of steps. This is important to note that the development of science as a whole...
programs, they being subject of transformation by the computer. Although a lot of techniques of processing programs (compilers, being, their creator. Although a lot of techniques of processing programs (compilers, macro-generators, verifies, etc.) were developed and put into practice during the four-decade history of computer science and engineering, only those which lead to deep transformation and are algorithmic in essence, are considered the main point of growth. Two such metacomputation methods were invented and are under further investigation now: supercompilation by V.Turchin and partial evaluation by the DIKU group led by N.Jones, the former being more general, the latter more specific, but more efficient.

In Section 2, Linguistic modeling and metacomputation, the role of the computer as a tool to execute linguistic models was discussed, and various metasystem control structures in activities of using the computer were revealed. Metacomputation as the automation of the activity of creating linguistic models, as well as the notation for MST by metacomputation, was introduced.

In section 3, Deriving linguistic models by metacomputation, we have shown that three main problems to be solved by metacomputation follow from the structure of the general schema of linguistic modeling introduced in Section 1. They are composition, inversion and specialization of programs. Composition and inversion are used to build a new model by abstracting the output from, and the input to, the original model, respectively. Specialization may be formally considered to be a special case of composition; however, it plays the role of the basis in constructing the methods of metacomputation. Two cases of the specialization are separated out, being basically treated by supercompilation and partial evaluation: defining a subset of arguments by an expression with variables and by fixing part of arguments, respectively.

In Section 4, Methods of metacomputation, supercompilation and partial evaluation was reviewed and compared. It has been shown that both of them have a clear metasystem structure consisting of a basic metacomputation technique that processes sets of data instead of concrete data, and a control that forces metacomputation to complete in finite time as well as performs some other functions. The two methods use two different ways of functioning of a control: on-line, by simultaneously observing and changing the activity to be controlled, and off-line, by preparing the control information in advance.

In Section 5, Manipulating language definitions, we have presented two the most dramatic tasks in linguistic modeling to be solved, or had been solved already, by metacomputation: converting compilative language definitions onto interpretive ones, and vice versa. It was demonstrated that while the former, building an interpreter from a compiler, requires program composition of general form to be effectively treated by metacomputation, the latter, building a compiler from an interpreter, reduces to a particular case of specialization. That's why it was first to be practically solved by the computer. The solutions of the both tasks are of our special interest, because they constructed of several MSTs performed by the same schema of metacomputation. They are examples of ultra-metasystem transition achieved by self-application of metacomputation. The feature of these series of MSTs is that they degenerate, that is, come to a fixed point, at the forth step, because the schema is formal and does not involve new information. The indeterminacy of the direction of an MST, even performed by the formal schema, has being illustrated as well.

In Section 6, Applications to mathematics, the first two subsections are devoted to the use of metacomputation in two rather different activities in mathematics: First was the application (with variations) of the schema of converting compilers into interpreters to the compressing hierarchies of mathematical definitions, which is needed to effectively process traditional mathematical texts by the computer. The second was a non-traditional approach to proving theorems by metacomputation, which uses the constructive representation of mathematical knowledge.

In the third subsection of Section 6 the new semantics of the classic mathematics developed by V.Turchin and called The Cybernetic Foundation was reviewed. It contains an MST in defining the central notion of mathematics -- the notion of truth. The Cybernetic Foundation as a whole has the structure of a ultra-MST consisting of a potentially infinite number of MSTs, each of which involves the user of the linguistic system to supply information about previous levels, which can't be evaluated algorithmically. Due to the creative energy of the user as a surrounding metasystem, although the MSTs are performed by the same schema they do not degenerate after a finite number of steps, and that's why the Cybernetic Foundation is free of Gödel
limitations.

We have demonstrated that in practical performing the MSTs that have already occured in computer science and mathematics, as well as that are to be performed by creative energy of scientics, the metasystem transition to metacomputation plays the crucial rule. This MST goes on now.

Acknowledgments

This work could not have been carried out without the pioneering work of Valentin Turchin and we are very grateful for many stimulating discussions on metasystem transition and related topics. The majority of the ideas of this paper are due to him. It is a pleasure to acknowledge fruitful discussions with the members of the Refal group. Special thanks are due to Sergei Abramov, Ruten Gurin, Victor Kistlerov, Arkady Klimov and Sergei Romanenko.

The third author of this paper was expected to be Alexander Romanenko. A hard illness and untimely departure did not allowed him to join us. The paper is devoted to his memory.

References


Turchin, V. F. (1972). (Equivalent transformations of recursive functions defined in Refal). Tjeorija Jazykov i Mjetody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods), Kiev-Alushta, USSR,


Turchin, V. F., A. V. Klimov, et al. (1977). *Bazisnyi Refal i yego realizatsiya na vychislitelnykh mashinakh (Basic Refal and its implementation on computers).* Moscow, GOSTROY SSSR, TsNIPIASS.