BC

Metasystem Transition Schemes in Computer Science and Mathematics

ROBERT GLÜCK‡ and ANDREI KLIMOV§

‡*Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark*
§*Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Miusskaja Sq. 4, RU-125047 Moscow, Russia*

(Received March 1, 1994; in revised form September 28, 1994; accepted May 30, 1995)

We analyze *metasystem transitions* which may be observed, or are intentionally used, in computer science and mathematics. Various metasystem structures are present in their activities of executing, creating and manipulating formal *linguistic models*. The crucial role in automating the creation and manipulation of linguistic models is played by *metacomputation*, that is, computation over formal models.

The manipulation of languages is one of the most essential problems of linguistic modeling. In this paper we analyze different schemes for transforming language definitions by metasystem transition and metacomputation, and present an example of *ultra-metasystem* transition. We show that *self-application* of metacomputation, a special case of metasystem transition, plays a central role in linguistic modeling. These techniques may also be utilized for reducing hierarchies of mathematical definitions and for manipulating mathematical texts effectively. Finally, we discuss a direct approach to theorem proving using a constructive representation of mathematical knowledge.

We derive the basic requirements for metacomputation by a structural analysis of different model definitions using a single concept, namely formal linguistic modeling, and show that three operations must be performed effectively and efficiently by metacomputation: *composition*, *inversion*, and *specialization* of linguistic models.

KEYWORDS: linguistic modeling, metacomputation, metasystem transition, self-application, program generation, theorem proving, program composition, program inversion, program specialization

1. INTRODUCTION

During the last decades we have witnessed tremendous technological breakthroughs in the development and application of computers. The introduction of the computer was a revolutionary step in the *execution* of formal linguistic models, a *metasystem transition* (Turchin, 1977). As a result, the number of linguistic models constructed and used has significantly increased. But this is not the last step! The next metasystem transition is to achieve control over the *creation* and *transformation* of linguistic models.

World Futures Vol. 45. pp. 213–243
Reprints available directly from the publisher
Photocopying permitted by license only

One of the defining features of modern science is the use of languages to construct *linguistic models* of reality. Informally, a *linguistic model* is a symbolic process which somehow mimics, or simulates, another primary process. Using a model enables the user, e.g. the human, to know something about a primary process before it actually happens, or without performing the primary process. For example, an engineer may create a mathematical model—a formal linguistic model—of a bridge in order to predict its stability and safety under various conditions. Virtually all areas of modern society rely on methods of formal linguistic modeling, a development which has been and will be intensified by the computer. The ground for this development has been prepared gradually during the last centuries by the introduction of formal methods in many areas of society, ranging from science and engineering to industry and business.

As does every branch of science, computer science and mathematics deal with their own type of objects, namely *formal linguistic models*, and the models of the models they create themselves. The computer was really necessary before one could start to learn more about formal linguistic modeling on a large scale (human beings are neither precise enough, nor fast enough to carry out any but the simplest procedures). Before the introduction of the computer, formal linguistic models were created, but had to be executed by hand and mind. Just as mastering the general principle of using physical tools gives rise to the creation of industrial systems, mastering the principle of linguistic modeling gives rise to the creation of hierarchical systems of languages and models.

The essence of *metacomputation* is to consider linguistic models as objects that can be transformed and manipulated mechanically (only mathematics created something comparable: metamathematics, which treats mathematical theories and proofs as formal objects). We say that the next step in formal linguistic modeling, the next large-scale metasystem transition, is achieved if efficient linguistic models can be created automatically by the computer and it suffices for the human to make initial formal definitions. Only when non-algorithmic specifications are involved, the human has to deal with them. The ultimate goal is to make the analysis and transformation of formal models by metacomputation a practical tool. Since the 60s many solutions have been tried in order to gain more control over the process of software development, some progress has been achieved, but the essential problems remain open.

In this paper we are especially interested in different schemes for metasystem transition and metacomputation, and their potential application in computer science and mathematics. Our aim is to show how metasystem transition over metacomputation may be applied to solve essential problems of linguistic modeling.

The paper is organized as follows: in Section 2 we introduce basic concepts of linguistic modeling, metacomputation and metasystem transition. Prerequisites for effective metacomputation are discussed in Section 3. In Section 4 we show how multiple metasystem transition may be used to transform compilative and interpretive language definitions into each other and we give an example of ultra-metasystem transition. Section 5 is devoted to applications in mathematics: processing mathematical texts by converting compilers into interpreters, and theorem proving by metacomputation.

2. LINGUISTIC MODELING AND METACOMPUTATION

In this section, we introduce the concepts of linguistic modeling, metacomputation and metasystem transition, as well as the basic scheme for creating models.

Computer as a Tool

The computer is a unique tool in the history of science and technology. Unlike earlier special-purpose machines, the computer is a universal symbol manipulating device that can run complex linguistic models accurately and fast (and with negligible power consumption). Linguistic models that can be executed by a computer, at least in principle, are referred to as *algorithms*, or *programs*.

The execution of a linguistic model is performed by a system consisting of two components: the computer hardware and the software. Each of the two components has its own impact on the evolution of linguistic modeling. Having different nature, hardware and software develop, to a large extent, independently, though they influence each other. Whereas advances in hardware technology result in quantitative changes (such as increasing processor speed, enlarging computer memory), the potential of software evolution is qualitative. In this paper we shall concern ourselves only with the latter.

Dimensions of Linguistic Modeling

We distinguish between two aspects of linguistic modeling (in both cases, *control* means the automation/mechanization of the respective activity):

control of model execution = computation
control of model creation = metacomputation

The introduction of the computer was a revolutionary step in the *execution* of formal linguistic models, a *metasystem transition* (MST). As a result the number of linguistic models created and used has significantly increased [in accordance with Turchin's law of branching growth of the penultimate level (Turchin, 1977)]. But the *creation* of linguistic models was not directly affected by the use of computers.

At the beginning, the creation of models was fully performed by the human ("programming"). Later, computer science tried to achieve more control over these activities by developing new paradigms for programming languages, constructing generators for special-purpose programs, and studying various approaches for the verification and transformation of programs. However, the basic problem still exists: how are we to achieve better control over the activity of creating formal linguistic models?

The approach discussed in this paper is to use linguistic modeling itself for the creation of new models and to repeatedly perform metasystem transitions over this process. We refer to the process of simulating analyzing and manipulating programs by programs as *metacomputation*, a term which underlines the fact that this activity is one level higher than ordinary computation (Fig. 1). From now on we will refer to formal linguistic models as linguistic models, or simply as models.

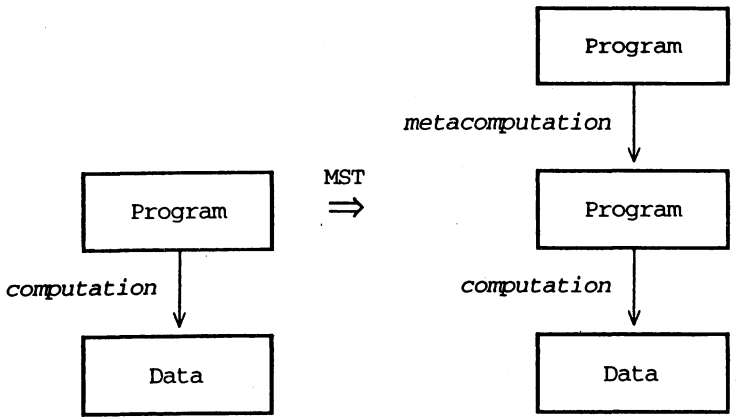

Figure 1. Metasystem transition from computation to metacomputation.

Computation

The execution of an algorithmic model, say P ("program") is performed by the computer hardware, say M ("machine"). The program P is written in the formal language M understood by the machine M. To denote the execution of a model P on a machine M with x ranging over all possible inputs we write

$$\langle P\, x\rangle_M \Rightarrow y$$

where, in order to emphasize the computation process, the result y is written after an arrow, rather than using an equality sign. Uppercase names denote constants (e.g., program P); lowercase names denote variables (e.g., variable x). We shall skip the language index M if it is clear from the context or not essential for the discussion.

Modeling

The basic method of modeling, often referred to as the *modeling scheme*, is as follows (Fig. 2). Let an object o be in some state, characterized by the information x_o, and suppose the object performs an action. The information about the ensuing state is denoted by $y_o = \langle F_o\, x_o\rangle$. (We do not assume that these actions are deterministic; $\langle F_o x_o\rangle$ denotes any of the possible states after an action.) Suppose we want to make a prediction about y_o. Modeling introduces another object m, a model, for making predictions about the object o, considering the model "similar," in a sense, to the object o. A model m is an abstraction of the object o, that is, a model contains less information than the object.

The mappings H_{in} and H_{out} are two abstraction functions that map the information x_o, y_o about the object o to the information x_m, y_m in the model m (since we consider only linguistic models, we can safely assume that the functions H_{in}, H_{out} do not interfere with the behavior of the object o). Having full information x_o, y_o, we can deduce the corresponding information x_m, y_m in the model, but not vice versa.

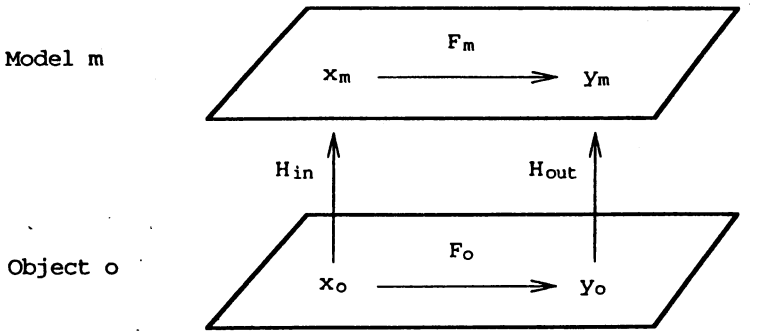

Figure 2. Modeling scheme.

Having information y_m, we cannot, in general predict an exact fact y_o about the object, but only that $\langle H_{out}\ y_o\rangle = y_m$. The predicted outcome belongs to the set $\{y_o \mid \langle H_{out}\ y_o\rangle = y_m\}$. Thus, by using F_m one can predict, to some extent, the state of the object resulting from F_o:

$$y_m = \langle F_m\ \langle H_{in}\ x_o\rangle\rangle = \langle H_{out}\ \langle F_o\ x_o\rangle\rangle \tag{2.1}$$

Creating Models from Models

Let the object o itself be a model and assume that we want to create a new model m that is a model of o. Initially we are given the object o, that is, a description of the domains X_o and Y_o over which x_o and y_o range, and a description of the function F_o: $X_o \to Y_o$. We need to define the domains X_m and Y_m which x_m and y_m range over, and the function F_m: $X_m \to Y_m$ of the new model m.

What can be automated in the definitions of a new model? The choice of the information used for building the new model is a creative step that depends on the goals of the user. Hence, we will not address the problem of *how* to choose mappings H_{in} and H_{out}; this choice is left to the user.

The main goal is to automate the construction of the new function F_m of model m. The statement (2.1), as well as the modeling scheme (Fig. 2), specify the function F_m: find some x_o such that $\langle H_{in}\ x_o\rangle = x_m$ (if it exists) and then compute y_m by evaluating $\langle H_{out}\ \langle F_o\ x_o\rangle\rangle$. We obtain the following definition of F_m using F_o, H_{in}^{-1} and H_{out} (where H_{in}^{-1} is an inverse of H_{in}):

$$\textbf{def}\ \langle F_m\ x_m\rangle = \langle H_{out}\ \langle F_o\ \langle H_{in}^{-1}\ x_m\rangle\rangle\rangle \tag{2.2}$$

We use the syntax **def** ... = ... for definitions. The function on the left hand side, F_m with the argument x_m, is defined by the expression on the right-hand side.

Metacomputation Schemes

Although the definition (2.2) fully specifies the new model F_m, in general it will not be efficient enough to execute it by the computer (or practically infeasible, as

often in the case of the inverse mapping H_{in}^{-1}). The goal of metacomputation is to derive new models which are "better" in some sense (e.g., more efficient with respect to run time and/or space consumption). We don't fix a specific set of metacomputation techniques at this level of discussion, as these may depend on the specific purpose and goal of metacompution. However, in section 3 we give three basic requirements for metacompution and in the sections 4 and 5 we show different applications that require techniques of different transformation power.

We now introduce a notation for metacomputation schemes. Let us denote the process of performing metacomputation by Mc. To express symbolically that metacomputation Mc is a metaactivity with respect to the model definition, we move the expression defining the model downwards (filling the remaining space with a line):

$$\langle \mathrm{Mc} \ \underline{\qquad\qquad\qquad} \rangle \Rightarrow F'_m \qquad (2.3)$$
$$\langle H_{out} \langle F_o \langle H_{in}^{-1} x_m \rangle \rangle \rangle$$

We say that the expression on the lower level is *metacoded*: the free variables in the original definition of the model (x_m in our case) become objects of the metacomputation process Mc (they are not replaced with values before the execution of Mc). The result of metacomputation is a program F'_m that is functionally equivalent to F_m: applying F'_m to a constant X produces the same result as F_m.

$$\langle F'_m \ X \rangle = \langle F_m \ X \rangle \qquad (2.4)$$

Performing metacomputation is a metasystem transition, and this is why formulas such as (2.3) are referred to as an *MST-formulas* [the two-dimensional notation was suggested by Turchin; see also (Glück, 1991)]. Note that a metacomputation process Mc is always applied to the definition (text) of a program rather than to its mathematical denotation (we consider linguistic models as our primary matter).

Multiple Metasystem Transition

A metaevaluator performing metacomputation, being itself just a formal linguistic model, may become the object of metacomputation. Since the number of such MST's is potentially unlimited, the repeated application of metacomputation may give rise to a hierarchy of metacomputation processes (Fig. 3). Consider metacomputing the expression $\langle F\ X,\ y \rangle$ where X is a constant and y a variable:

$$\langle \mathrm{Mc} \ \underline{\qquad\qquad} \rangle \Rightarrow F_x \qquad (2.5)$$
$$\langle F\ X,\ y \rangle$$

Assume that we want to define the metacomputation process (2.4) for any constant (not only for constant X). That is, we want to replace the constant X with a free variable x. Since a *variable* under the top level (e.g., y) is an object of the metaevaluator Mc rather than a parameter of the formula, we have to raise the *constant* X to the top level in the MST-formula, denoting its original position by a bullet •.

Note that moving constants up and down in an MST-formula does not change the result as long as their original position is clearly identified. The expression on the left side of (2.6) is read as follows: take constant X and metacode it once before

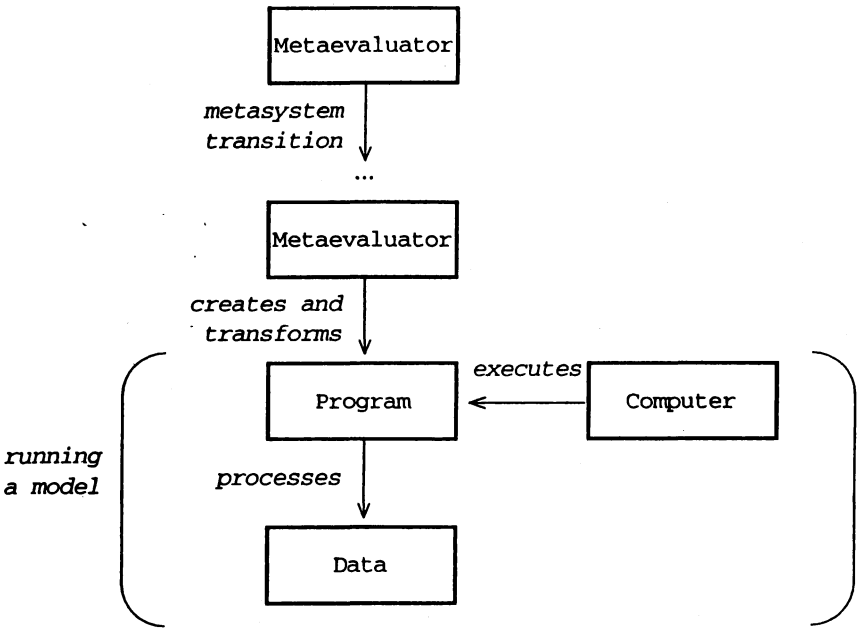

Figure 3. Multiple metasystem transitions over programs.

moving it down to its original position; the right side denotes the same: constant X is metacoded once.

$$\langle \mathtt{Mc}\ \underset{\langle \mathtt{F}\ \bullet, \mathtt{y}\rangle}{\mathtt{X}}\ \rangle = \langle \mathtt{Mc}\ \underset{\langle \mathtt{F}\ \mathtt{X}, \mathtt{y}\rangle}{}\ \rangle \tag{2.6}$$

The class of computation processes *parameterized* by x is then defined by replacing constant X on the top level by variable x. Note that the variables x and y belong to different levels: x is free on the top level, while y is an object for Mc.

$$\mathbf{def}\ \langle \mathtt{McF}\ \mathtt{x}\rangle = \langle \mathtt{Mc}\ \underset{\langle \mathtt{F}\ \bullet, \mathtt{y}\rangle}{\mathtt{x}}\ \rangle \tag{2.7}$$

Variables that are raised from their original position, such as x, are called *elevated.* To analyze and manipulate the metacomputation process defined by McF, we make it the object of another metacomputation process, say Mc′ (thereby we obtain a three-level MST-formula). The result is a program McF′ which is functionally equivalent to McF.

$$\langle \mathtt{Mc}'\ \underset{\langle \mathtt{McF}\ \mathtt{x}\rangle}{}\ \rangle = \langle \mathtt{Mc}'\ \underset{\langle \mathtt{Mc}\ \underset{\langle \mathtt{F}\ \bullet, \mathtt{y}\rangle}{\mathtt{x}}\ \rangle}{}\ \rangle \Rightarrow \mathtt{McF}' \tag{2.8}$$

If the metaevaluator Mc′ is identical to the metacomputed system Mc, that is, Mc′ = Mc, we speak of *self-application.*

3. SELECTED PROBLEMS OF METACOMPUTATION

Starting from a single concept, namely linguistic modeling, we consider different possible model definitions and determine three basic operations on linguistic models (Glück and Klimov, 1994): *composition*, *inversion* and *specialization* (the latter being a special, though important, case of composition). Various program transformation paradigms in computer science are motivated by one or more of these metacomputation tasks.

3.1. Problem of Program Composition

The problem of program composition arises when we abstract only the output in the model definition. Consider the case when the information x_o on the object model o is identical to the information x_m on the new model m (Fig. 4). That is, the mapping H_{in} is the identity function: $\langle H_{in}\ x_o\rangle = x_o$. The function of the new model is then defined as

$$\textbf{def}\ \langle F_m\ x_m\rangle = \langle H_{out}\ \langle F_o\ x_m\rangle\rangle$$

Assume that we are interested in a small part of the output y_o. To define the new model, we provide the mapping H_{out} selecting the parts of y_o we are interested in. However, directly computing the above definition of the new model does not decrease the amount of computer resources needed to obtain the result y_m. In this case it might pay off to create a new model by metacomputation:

$$\langle \text{Mc}\ \frac{\qquad}{\langle H_{out}\ \langle F_o\ x\rangle\rangle}\rangle \Rightarrow F'_m$$

One would expect that redundant computations in F_o are removed during metacomputation, and only those computations that are needed to produce the information selected by H_{out} are present in F_m. The importance of effectively metacomputing the composition of programs is hard to overestimate. Composition is one of the main methods of constructing complex programs from simpler ones. This is why *supercompilation* (Turchin, 1986), a powerful method for metacomputation, was directed to the problem of program composition from the beginning. The derivation of programs by abstracting output information has been studied in connection with *program slicing* (Weiser, 1984).

Below we shall consider particular applications of program composition: the conversion of compilers to interpreters (Section 4.1) and the reduction of hierarchies of mathematical definitions (Section 5.1). A special, but important case of program composition is program specialization (Section 3.3).

3.2. Problem of Program Inversion

The problem of program inversion arises when we abstract only the input in the model definition. Consider the case of deriving a new model when the output

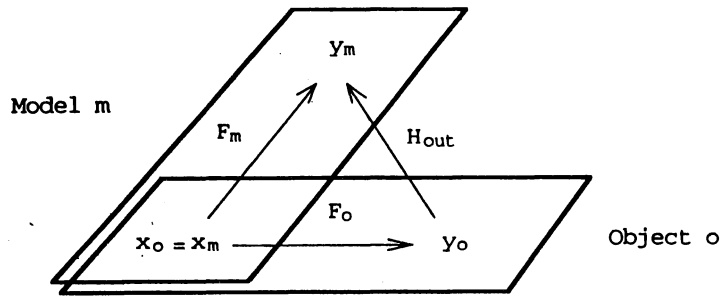

Figure 4. Deriving a model by abstracting only the output information.

information y_o on the object model is identical to the output information y_m on the new model (Fig. 5). That is, mapping H_{out} is the identity function: $\langle H_{out}\ y_o\rangle = y_o$. The function F_m of the model is then defined as follows:

$$\textbf{def}\ \langle F_m\ x_m\rangle = \langle F_o\ \langle H_{in}^{-1}\ x_m\rangle\rangle$$

As in the case of composition, one may use metacomputation to derive a new model F'_m that is functionally equivalent to F_m, but potentially more efficient:

$$\langle \text{Mc}\ \frac{}{\langle F_o\ \langle H_{in}^{-1}\ x_m\rangle\rangle}\rangle \Rightarrow F'_m$$

In this case a combination of composition and inversion is used. We want to derive a new model that can be used to make a prediction y_m using the partial information x_m about x_o. The mapping H_{in} defines what is to be known about x_o: $x_m = \langle H_{in}\ x_o\rangle$. In some instances, the partial information x_m will be sufficient to produce y_m. Then $\langle F_m\ x\rangle$ should return it. However, generally x_m does not define y_m exactly. In this case, one is interested either in an existential solution (one of the possible results), or in an universal solution (all possible results). These two choices result in two different types of inversions.

The problem of constructing an inverse relation is interesting in its own right, since many mathematical problems are formulated in the following way: given the

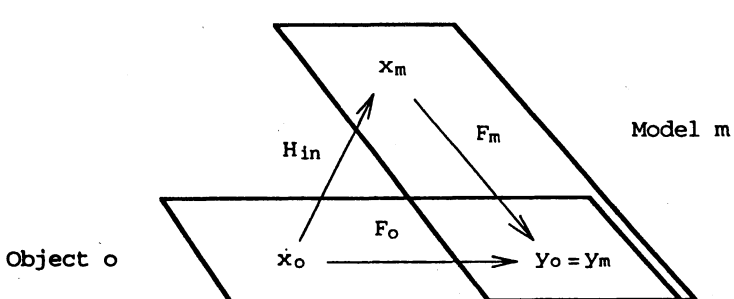

Figure 5. Deriving a model by abstracting only the input information.

description P of the properties of an object, find (at least one) object x such that ⟨P x⟩ = True. This is referred to as the *inversion problem.*

The inversion of programs is a fundamental problem, and a large branch of computer science, *logic programming,* has been based on solutions emerging from logic and proof theory (Kowalski, 1979). Logic programming may be considered as a particular case of inverse problem solving (Abramov, 1991). Direct methods for inverting algorithms have also been developed (Turchin, 1972; Harrison, 1988; Romanenko, 1991).

Inversion Solver as a Metasystem

Let Search be a program that takes the definition of a model F and an output value Y (e.g., True) and executes ⟨F x⟩ for all possible input values. (We deal with linguistic, rather than set-theoretical objects. Since the former are denumerable, the process of running through all possible objects is well defined.) The program stops if and only if for some x: ⟨F x⟩ = Y and returns the first value found for x. This is a simple, but rather inefficient method for solving the inversion problem. The program Search is actually a metasystem over models. It takes a model F as input and performs some form of control over its execution. This is expressed by the MST-formula

$$\langle \texttt{Search} \ \frac{\qquad\qquad}{\langle \texttt{Equal} \ \langle \texttt{F x} \rangle, \texttt{Y} \rangle} \rangle$$

where Equal tests the equality of two values. Note that F and Y are constants and that x is a variable whose value is sought. A simple, but universal algorithm for solving the inversion problem, i.e. for an arbitrary model and any output, is then defined as

$$\textbf{def} \ \langle \texttt{Solve f y} \rangle = \langle \texttt{Search} \ \frac{\qquad\texttt{f}\qquad\texttt{y}}{\langle \texttt{Equal} \ \langle \bullet \ \texttt{x} \rangle, \bullet \rangle} \rangle$$

Given an algorithm solving the inversion problem, e.g. Solve, one may produce the inverse model F^{-1} by metacomputing Solve with respect to a given F:

$$\langle \texttt{Mc} \ \frac{\qquad\qquad}{\langle \texttt{Solve F,y} \rangle} \rangle \Rightarrow F^{-1}$$

Similar MST schemes can be defined for other solvers that are not so trivial as the one defined above. Different inverse models F^{-1} may be generated from the same model F by varying the solver Solve and/or the metaevaluator Mc.

3.3. Problem of Program Specialization

Automatic inversion by metacomputation is a hard problem, and hence it is important to consider special cases of the modeling scheme not requiring it. This leads to the problem of program specialization. Instead of solving the inversion problem, the inverse mapping, say G, is supplied by the user: $\langle G \ x_m \rangle = \langle H_{in}^{-1} \ x_m \rangle$. This corresponds to changing the direction of the arrow labeled with H_{in} (cf. Fig. 5 and Fig. 6). The function F_m of the new model is then defined as

$$\textbf{def} \ \langle F_m \ x_m \rangle = \langle F_o \ \langle G \ x_m \rangle \rangle$$

The metacomputation problem in this definition has the same structure as the problem of composition (Section 3.1). However, since the modeling function F_o is usually much more complex than the mappings H_{out}, H_{in}, and G, there is a difference between the two cases: in Section 3.1, the outer function H_{out} is simpler, and here, the inner function G is simpler. Hence, different methods of metacomputation may be advantageous in each case. The problem of program specialization falls into the second class.

Specialization

The problem of specialization arises when the domain for which a model o is used is restricted. The function of the specialized model is the same as the function F_o of the original model o, but its input ranges only over a part of the original domain. Having defined the specialized model, we can metacompute its definition to remove redundant computations which may be present in the original model F_o but are not necessary for computations within the narrowed domain S. In fact, any function G defines some restricted input domain $S \in X_o$:

$$S = \{\langle G\ x_m\rangle \mid x_m \in X_m\}$$

However, restricted forms of G are preferred in practice in order to simplify the task of metacomputation. The function G can be defined as *filter* (Turchin, 1980a), i.e. as the identity function for all elements of S and undefined otherwise:

def ⟨G x⟩ = x **if** x ∈ S, undefined otherwise

Another example is the specialization of F_o with respect to parameterized, nonrecursive descriptions of the subset S. Consider a subset S containing triples of the form [A, x, x], where the first component is a constant A and the second and third component are identical. Then the function G can be defined as

def ⟨G x⟩ = [A, x, x]

Having identified the problem of specialization as a case of program composition, we can apply metacomputation methods developed for program composition and perform specialization of F_o with respect to the subset defined by G as follows:

$$\langle Mc\ \frac{\qquad}{\langle F_o\ \langle G\ x_m\rangle\rangle}\rangle \Rightarrow F'_m$$

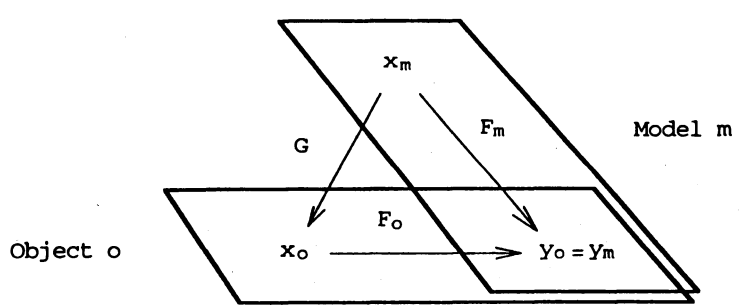

Figure 6. Deriving a model by restricting the input information.

Partial Evaluation

The problem of specialization can be simplified even more if the input of the original model o consists of several arguments and some of the arguments are fixed to constants. Let F_o be a function of three arguments, e.g. $\langle F_o\ x_1,\ x_2,\ x_3\rangle$. Assume that the first argument is equal to a constant A. Then the specialized model is defined as

$$\textbf{def}\ \langle F_m\ x_2, x_3\rangle = \langle F_o\ A, x_2, x_3\rangle$$

A more efficient model F_m can be obtained by metacomputing the definition of the specialized model.

$$\langle \mathrm{Mc}\ \underline{\qquad\qquad}\rangle \Rightarrow F'_m$$
$$\langle F_o\ A, x_2, x_3\rangle$$

This can give substantial savings if a significant part of F's computations depends only on the first argument.

To show the relation between metacomputation and partial evaluation, we define a program specializer through a metaevaluator. Let Spec be a program specializer with two arguments, a program p and data x. Given the program p and part of its input data x, the specializer Spec produces a specialized program p_x that, when applied to the remaining input y, returns the same result as the original program p when applied to x and y:

$$\langle \mathrm{Spec}\ p, x\rangle \Rightarrow p_x \text{ such that } \langle p_x\ y\rangle = \langle p\ x, y\rangle$$

Then the program specializer Spec can be defined through the metaevaluator Mc:

$$\textbf{def}\ \langle \mathrm{Spec}\ p, x\rangle = \langle \mathrm{Mc}\ \underline{p\ x}\ \rangle$$
$$\langle \bullet\ \bullet, y\rangle$$

This case of program specialization gave rise to a research area called *partial evaluation* which has been developing rapidly during the last decade (Jones et al., 1993). Surprisingly many problems in computer science can be reduced to partial evaluation, including the central problem of metacomputation: the problem of *self-applying* metacomputation. It was found (Futamura, 1971) that the solution to the problem of generating compilers from interpreters requires neither composition, nor inversion of programs, just fixing some of the arguments is sufficient (Section 4.2). Although the problem of specialization is as a special case of program composition, it is worth considering it separately, because of the large number of practical problems that require only methods of program specialization.

4. MANIPULATING LANGUAGE DEFINITIONS

Languages and their definition play a central role in all forms of linguistic modeling. In this section we consider two basic schemes for converting formal language definitions into each other, namely interpretive to compilative definitions, and vice versa. Upon introducing interpretive and compilative language definitions,

we show how metacomputation and multiple MST can be used for their transformation, and conclude with an example of ultra-MST.

Languages and Linguistic Modeling

A characteristic feature of linguistic modeling is that one deals with a variety of languages created for a wide range of applications. As new problems arise, existing languages are modified and new languages are created. A language suited for one problem is not necessarily the best for another problem and we need to be able to choose whatever formalism is adequate for describing and solving a problem. But one can never be sure of finding what is needed among those formalisms that have already been applied to previous problems (e.g., when Newton found his law of gravitation, he needed to invent the differential calculus and the concept of the second derivative). We conclude this short discussion in the hope the reader has been convinced that the ability to define and freely manipulate language definitions is a necessary prerequisites for effective linguistic modeling.

Language Definitions

For a language to have semantics, meaning, there must exist an agent—be it a machine or the human—that understands the language. This understanding is demonstrated by performing activities expressed ("controlled") by the language. If the available languages are expressive enough, one may start defining new languages in terms of existing ones. Thus, the world of languages can be extended by its own means. This gives rise to hierarchical systems of formal languages and linguistic models. Executable languages form the foundation of this world of languages. We know exactly two forms of language definitions: compilative and interpretative

- *Compilation* is the process of translating expressions from one language, say N, to another language, say L. The new language N is defined in terms of the language L where `n` and `l` are expressions in N and L, respectively:

$$\langle \texttt{Comp n} \rangle_{\texttt{M}} \Rightarrow \texttt{l}$$

The languages N and L are referred to as *source* and *target languages*, respectively. The compilation is defined in the language M, the *metalanguage*.

The result of the activity performed by an agent that understands language L is the same, when given the target program `l`, as that of an agent that understands language N. when given the corresponding source program `n`:

$$\langle \texttt{n x} \rangle_{\texttt{N}} = \langle \langle \texttt{Comp n} \rangle_{\texttt{M}} \texttt{x} \rangle_{\texttt{L}}$$

- *Interpretation* amounts to performing the activity implied by a program, say `n`, using another universal program, say `Int`, called the interpreter.

$$\langle \texttt{n x} \rangle_{\texttt{N}} = \langle \texttt{Int n,x} \rangle_{\texttt{M}}$$

The result of the activity performed by the interpreter `Int` is the same as the result returned by an agent that understands the language N directly. The source language N is defined directly through actions in the metalanguage M.

Converting Language Definitions

Generally speaking, the need for converting compilers into interpreters may arise when a particular language is gradually extended. In this case it may be easier to define the extensions by a translation into simpler expressions than by an interpretive definition. However, when a new language is defined from scratch, it is usually easier to define the language by means of an interpreter rather than a compiler. In this case, the opposite problem may arise: the conversion of an interpreter to a compiler.

It is interesting to note that mathematics mainly deals with the former case, while in computer science the latter is more frequent. In Section 5.1 we shall return to this topic and demonstrate the need for converting compilative definitions into interpretive definitions in mathematics. Now we discuss both problems from the perspective of computer science, because some constructive results have already been achieved in computational practice.

4.1. Converting Compilers to Interpreters

Let `CompNL` be an N→L-compiler and let `IntL` be an L-interpreter. Then the execution of an N-program can be performed in two steps: (i) translate the N-program into language L using the N→L-compiler; (ii) interpret the new L-program with the L-interpreter:

$$\langle \texttt{IntL}\ \langle \texttt{CompNL}\ \texttt{n}\rangle, \texttt{x}\rangle \Rightarrow \texttt{y}$$

A new N-interpreter, `IntN`, can be defined as

$$\textbf{def}\ \langle \texttt{IntN}\ \texttt{p}, \texttt{x}\rangle = \langle \texttt{IntL}\ \langle \texttt{CompNL}\ \texttt{p}\rangle, \texttt{x}\rangle \qquad (4.1.1)$$

However, the step via an intermediate language is sometimes rather inefficient (in Section 5.1 we shall see an example where compilation before interpretation is practically impossible).

First MST

A more efficient N-interpreter `IntN'` may be obtained by metacomputing the definition (4.1.1), that is, by performing an MST over the composition of `IntL` and `CompNL`.

$$\langle \texttt{Mc}\ \underbrace{\qquad\qquad\qquad\qquad}_{\langle \texttt{IntL}\ \langle \texttt{CompNL}\ \texttt{p}\rangle, \texttt{x}\rangle}\rangle \Rightarrow \texttt{IntN}' \qquad (4.1.2)$$

This MST-formula defines the process of generating an N-interpreter `IntN'` from the N→L-compiler `CompNL`. Since there are two variables, `p` and `x`, below the level of the metaevaluator `Mc`, the generated program `IntN'` accepts two arguments:

$$\langle \texttt{IntN}'\ \texttt{n}, \texttt{x}\rangle \Rightarrow \texttt{y}$$

Second MST

To define the metacomputation process (4.1.2) for compilers with different source languages, we replace the compiler CompNL with a variable, say comp, and lift the variable to the top level where it becomes free (recall that if a variable is at the level under the metacomputation process Mc, it becomes an object of that process):

$$\textbf{def}\ \langle \texttt{IntGen comp} \rangle = \langle \texttt{Mc}\ \underset{\langle \texttt{IntL} \langle\ \bullet\ \ \texttt{p} \rangle, \texttt{x} \rangle}{\texttt{\_\_\_\_\_comp\_\_\_\_\_}} \rangle \qquad (4.1.3)$$

This MST-formula defines the process of generating an N-interpreter from an N→L-compiler, where N is an arbitrary source language and L is the fixed target language. To make this process more efficient we metacompute its definition:

$$\langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \underset{\langle \texttt{IntL} \langle\ \bullet\ \ \texttt{p} \rangle, \texttt{x} \rangle}{\texttt{\_\_\_\_\_comp\_\_\_\_\_}} \rangle}{\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}} \rangle \Rightarrow \texttt{IntGen}' \qquad (4.1.4)$$

The result of this two-level MST-formula is a program whose arguments are determined by the variables below the level of the top metaevaluator Mc. Since there is only one variable at this level, comp, the generated program IntGen′ accepts one argument (i.e., a compiler). From the next lower level we can determine that the result of executing IntGen′ is a program which takes two arguments (i.e., p and x). The new program performs the function defined through the program IntL. Therefore, the program IntGen′ can be regarded as interpreter generator:

$$\langle \texttt{IntGen}'\ \texttt{CompNL} \rangle \Rightarrow \texttt{IntN}'$$

Third MST

To define the metacomputation process (4.1.4) for compilers with target languages different from L, we make one more MST. First, we replace the L-interpreter IntL with a variable, say int:

$$\textbf{def}\ \langle \texttt{IntGenGen int} \rangle = \langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \underset{\langle\ \bullet\ \langle\ \bullet\ \ \texttt{p} \rangle, \texttt{x} \rangle}{\texttt{\_\_\|\_\_comp\_\_\_\_\_}} \rangle}{\texttt{\_\_\_\_int\_\_\_\_\_\_\_\_\_\_\_\_}} \rangle \qquad (4.1.5)$$

This MST-formula defines the process of producing interpreter generators for different target languages. Second, we metacompute the definition to make the defined process more efficient:

$$\langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \underset{\langle\ \bullet\ \langle\ \bullet\ \ \texttt{p} \rangle, \texttt{x} \rangle}{\texttt{\_\_\|\_\_comp\_\_\_\_\_}} \rangle}{\texttt{\_\_\_\_int\_\_\_\_\_\_\_\_\_\_\_\_}} \rangle}{\texttt{\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}} \rangle \Rightarrow \texttt{IntGenGen}' \qquad (4.1.6)$$

The result of this three-level MST-formula is a program IntGenGen′ that takes as argument an interpreter (see variable int under the top level). From the next lower level we can determine that the result of IntGenGen′ is a program which takes one argument, a compiler comp, which in turn produces a program whose arguments

are determined by the variables at the next lower level, p and x. Therefore, the program IntGenGen′ can be regarded as a generator of interpreter generators:

$$\langle \texttt{IntGenGen IntL} \rangle \Rightarrow \texttt{IntGen}'$$

4.2. Converting Interpreters to Compilers

Let IntN be an N-interpreter written in the language M. Given an M-machine, an N-program P_N can be executed using the N-interpreter IntN, where x stands for some input for the program P_N:

$$\textbf{def}\ \langle P_M\ x \rangle = \langle \texttt{IntN}\ P_N, x \rangle_M \tag{4.2.1}$$

However, the repeated interpretation of the same N-program P_N with varying input may be rather inefficient.

First MST

A more efficient version of the N-program P_N may be obtained, if we compile it into an equivalent M-program that can be executed directly by the M-machine. This can be achieved by metacomputing the definition (4.2.1):

$$\langle \texttt{Mc} \ \underset{\langle \texttt{IntN}\ P_N, x \rangle}{\_\_\_\_\_} \rangle \Rightarrow P'_M \tag{4.2.2}$$

This MST-formula defines the compilation of the N-program P_N to an M-program P'_M. The new program P'_M is equivalent to P_N, but often much more efficient because it can be executed directly by the M-machine:

$$\langle P_M\ x \rangle_M = \langle \texttt{IntN}\ P_N, x \rangle_M$$

Second MST

To define the metacomputation process (4.2.2) for arbitrary N-programs, rather than for a particular N-program P_N, we replace P_N with a variable, say p, and lift it to the top-level:

$$\textbf{def}\ \langle \texttt{CompNL p} \rangle = \langle \texttt{Mc} \ \underset{\langle \texttt{IntN} \bullet, x \rangle}{\_\_\_\_\texttt{p}\_\_\_\_} \rangle \tag{4.2.3}$$

This MST-formula defines the process of compiling N-programs into M-programs. To make this process more efficient we metacompute its definition:

$$\langle \texttt{Mc} \ \underset{\langle \texttt{Mc} \ \underset{\langle \texttt{IntN} \bullet, x \rangle}{\_\_\_\_\texttt{p}\_\_\_\_} \rangle}{\_\_\_\_\_} \rangle \Rightarrow \texttt{CompNL}' \tag{4.2.4}$$

The result, CompNL′, is a program which, given an N-program P_N, returns an equivalent program written in the language M. Therefore CompNL′ can be regarded as an N→M-compiler:

$$\langle \texttt{CompNL}'\ P_N \rangle \Rightarrow P_M$$

Third MST

To define the metacomputation process (4.2.4) for any interpreter, rather than for a particular `IntN`, we replace the interpreter `IntN` with a variable, say `int`, and make one more MST:

$$\textbf{def}\ \langle \texttt{CoGen int} \rangle = \langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \_\mid\_\texttt{p}\_\_\_\rangle}{\underset{\langle \bullet\ \ \bullet, \texttt{x} \rangle}{\_\_\_\_\texttt{int}\_\_\_\_\_}} \rangle \qquad (4.2.5)$$

This MST-formula defines the process of generating N→M-compilers from N-interpreters. To make this process more efficient we metacompute its definition:

$$\langle \texttt{Mc}\ \underset{\langle \texttt{Mc}\ \_\_\_\_\texttt{int}\_\_\_\_\_\rangle}{\underset{\langle \texttt{Mc}\ \_\mid\_\texttt{p}\_\_\_\rangle}{\underset{\langle \bullet\ \ \bullet, \texttt{x} \rangle}{\_\_\_\_\_}}} \rangle \Rightarrow \texttt{CoGen}' \qquad (4.2.6)$$

The result, `CoGen'`, is a program which, when applied to an interpreter `IntN` for some language N, returns an N→M-compiler. The MST-formula (4.2.6) can be read from top to bottom: `CoGen'` is a program, which takes one argument (variable `int`) and produces a program, which takes one argument (variable `p`) and produces a program, which takes one argument (variable `x`) and produces a result equivalent to ⟨`int p,x`⟩. Therefore the program `CoGen'` can be regarded as a compiler generator:

$$\langle \texttt{CoGen' IntN} \rangle \Rightarrow \texttt{CompNM}$$

Reduction to Partial Evaluation

Each of the metacomputation layers in the formula (4.2.6) fits the format of partial evaluation. That is, each of the three MSTs can be expressed by using the specializer `Spec` (Section 3.3):

$$\textbf{def}\ \langle \texttt{Spec p,x} \rangle = \langle \texttt{Mc}\ \underset{\langle \bullet\ \ \bullet, \texttt{y} \rangle}{\_\texttt{p}\_\texttt{x}\_\_\_} \rangle$$

Initial formula: $\langle \texttt{IntN}\ P_N, X \rangle \Rightarrow Y$

First MST: $\langle \texttt{Mc}\ \underset{\langle \texttt{IntN}\ P_N, \texttt{x} \rangle}{\_\_\_\_\_} \rangle = \langle \texttt{Spec IntN}, P_N \rangle \Rightarrow P'_M$

Second MST: $\langle \texttt{Mc}\ \underset{\langle \texttt{Spec IntN,p} \rangle}{\_\_\_\_\_} \rangle = \langle \texttt{Spec Spec, IntN} \rangle \Rightarrow \texttt{CompNL}'$

Third MST: $\langle \texttt{Mc}\ \underset{\langle \texttt{Spec Spec, int} \rangle}{\_\_\_\_\_} \rangle = \langle \texttt{Spec Spec, Spec} \rangle \Rightarrow \texttt{CoGen}'$

The same principle—replacing the value of the second argument by a variable and metacomputing the new expression—can be used to perform the fourth MST. However, the fourth MST reproduces the `CoGen'` from the third MST, as pointed out in (Futamura, 1983):

Fourth MST: $\langle \texttt{Mc}\ \underset{\langle \texttt{Spec Spec, spec} \rangle}{\_\_\_\_\_} \rangle = \langle \texttt{Spec Spec, Spec} \rangle \Rightarrow \texttt{CoGen}'$

This is an example of a degeneration of a series of MSTs which is performed according to a formalized scheme without adding new knowledge. In the next section, we shall discuss this effect in more detail and show how to avoid degeneration by supplying new knowledge.

Conclusion

We should stress that the MST-formulas assert nothing about the quality of the generated programs. The efficiency of the generated programs depends on the particular metacomputation techniques applied by the metaevaluator. The MST-formulas are high-level schemes for formulating and defining metacomputation tasks.

The conversion of interpreters to compilers was first described in (Futamura, 1971). These formulas are known as *Futamura projections.* The first compilers according to this MST scheme were produced using partial evaluation techniques (Jones *et al.*, 1985). Many problems of compiler construction are not solved by these MST-formulas, e.g. the problem of code generation. But it is a noteworthy step towards the automatic construction of compilers and the transformation of interpretive programs.

Recently an approach to improve program transformation by inserting interpreters in MST-formulas has been developed (Turchin, 1993; Glück and Jørgensen, 1994). Related work also includes MST-schemes for multiple and incremental metasystem transition (Glück, 1991) and an MST-scheme for generating program specializers from interpreters (Glück, 1994). The difficulty in principle of self-applying metaevaluators was identified as a consequence of Ashby's law of requisite variety in (Glück, 1992).

4.3. An Ultra-Metasystem Transition

An *ultra-metasystem* is a broader system which provides and maintains conditions for a series of MSTs (Turchin, 1977). If we are able to formalize a system that possesses the internal potential for development by MST, then we can perform an ultra-MST due to the *staircase effect.* In the previous sections we showed how to perform a series of MSTs where each successive MST was formed from the preceding one according to a certain scheme. When such a scheme is established, new MSTs can be performed mechanically. In this section we consider an example of ultra-MST based on the scheme developed in the previous sections. We use this ultra-MST to illustrate an interesting effect of *degeneration of a series of MSTs,* when, after some MST is performed, no new system is generated.

The MST series in the previous sections are constructed according to the same procedure. They start with the definition of an initial computation process, e.g. ⟨IntL ⟨CompNL p⟩, x⟩ (Section 4.1), or ⟨IntN p_N, x⟩ (Section 4.2), and then MSTs are performed repeatedly.

MST Scheme

(0) Start with a formula defining an initial process.
(1) Define a new parameterized formula by replacing some parts of the given formula with variables $x_1 \ldots x_n$ (in order to express that the variables are free, they are lifted to the top level). The parameterized formula defines a class of computation processes.
(2) Add a new metacomputation level over the parameterized formula (denoted by ⟨Mc ____⟩). The free variables now become objects of the metacomputation process. The result of metacomputation is a program that accepts arguments corresponding to the variables $x_1 \ldots x_n$ and has the same function as the original process defined in (1).
(3) The steps (1) and (2) can be repeated until all constant parts of the initial formula have been replaced with variables.

Indeed, the three MSTs (Sections 4.1 and 4.2) abstract step by step all parts of the basic formula. In principle, one could choose another order of introducing variables in the MSTs, but in our case these variations have only minor significance.

Can further MSTs be performed? Having formulated the above MST scheme which controls a series of MSTs, we can try to perform an ultrametasystem transition. Note that at each MST step of the above procedure a new constant, the metaevaluator Mc, is added which may be replaced with a variable in one of the next steps: an MST over metaevaluators is performed.

Thus, the fourth MST formula (4.3.1) can be defined by replacing the innermost metaevaluator in (4.2.6) with a variable, say mc′, and adding a new metacomputation level. The result of metacomputing the new formula is a compiler-generator generator CoGenG that produces a compiler-generators given a metaevaluator. The fifth MST formula (4.3.2) is obtained from the fourth MST by replacing the next metaevaluator with a variable, say mc″. The result is a compiler generator CoGenGG that generates a CoGenG given a metaevaluator; and so forth.

Fourth MST

```
⟨Mc ____________________________⟩ ⇒ CoGenG        (4.3.1)
   ⟨Mc ____ mc' ____________⟩
      ⟨Mc __|__ int _____⟩
         ⟨ • __|__ p ___⟩
            ⟨ •  •,x⟩
```

Fifth MST

```
⟨Mc ____________________________⟩ ⇒ CoGenGG       (4.3.2)
   ⟨Mc ____ mc" ____________⟩
      ⟨Mc __|__ mc' ____________⟩
         ⟨ • __|__ int _____⟩
            ⟨ • __|__ p ___⟩
               ⟨ •  •,x⟩
```

Sixth MST

$$\langle \mathtt{Mc}\ \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\rangle \Rightarrow \mathtt{CoGenGGG} \qquad (4.3.3)$$
$$\langle \mathtt{Mc}\ \underline{\quad \mathtt{mc'''} \quad\qquad\qquad\qquad\qquad}\rangle$$
$$\langle \mathtt{Mc}\ \underline{\quad|\quad \mathtt{mc''} \qquad\qquad\qquad\qquad}\rangle$$
$$\langle\ \bullet\ \underline{\quad|\quad \mathtt{mc'} \qquad\qquad\qquad}\rangle$$
$$\langle\ \bullet\ \underline{\quad|\quad \mathtt{int} \qquad\qquad}\rangle$$
$$\langle\ \bullet\ \underline{\quad|\quad \mathtt{p} \quad}\rangle$$
$$\langle\ \bullet\ \ \bullet, \mathtt{x}\rangle$$

Applying CoGenG, CoGenGG, and CoGenGGG to a metaevaluator Mc′ different from Mc, produces the following new compiler generators:

$$\langle \mathtt{CoGenG\ Mc'}\rangle \Rightarrow \mathtt{CoGen'} \qquad (4.3.4)$$
$$\langle\langle \mathtt{CoGenGG\ Mc'}\rangle\ \mathtt{Mc'}\rangle \Rightarrow \mathtt{CoGen''} \qquad (4.3.5)$$
$$\langle\langle\langle \mathtt{CoGenGGG\ Mc'}\rangle\ \mathtt{Mc'}\rangle\ \mathtt{Mc'}\rangle \Rightarrow \mathtt{CoGen'''} \qquad (4.3.6)$$

What is their difference? This can easily be seen after substituting Mc′ into the definitions of CoGenG, CoGenGG, and CoGenGGG (4.3.1–3):

$$\langle \mathtt{Mc}\ \underline{\qquad\qquad\qquad\qquad}\rangle = \mathtt{CoGen'} \qquad (4.3.7)$$
$$\langle \mathtt{Mc}\ \underline{\quad \mathtt{int} \quad}\rangle$$
$$\langle \mathtt{Mc'}\ \underline{\quad|\quad \mathtt{p} \quad}\rangle$$
$$\langle\ \bullet\ \ \bullet, \mathtt{x}\rangle$$

$$\langle \mathtt{Mc}\ \underline{\qquad\qquad\qquad\qquad}\rangle = \mathtt{CoGen''} \qquad (4.3.8)$$
$$\langle \mathtt{Mc'}\ \underline{\quad \mathtt{int} \quad}\rangle$$
$$\langle \mathtt{Mc'}\ \underline{\quad|\quad \mathtt{p} \quad}\rangle$$
$$\langle\ \bullet\ \ \bullet, \mathtt{x}\rangle$$

$$\langle \mathtt{Mc'}\ \underline{\qquad\qquad\qquad\qquad}\rangle = \mathtt{CoGen'''} \qquad (4.3.9)$$
$$\langle \mathtt{Mc'}\ \underline{\quad \mathtt{int} \quad}\rangle$$
$$\langle \mathtt{Mc'}\ \underline{\quad|\quad \mathtt{p} \quad}\rangle$$
$$\langle\ \bullet\ \ \bullet, \mathtt{x}\rangle$$

Provided that the new metaevaluator Mc′ is "better," in some sense, than the original metaevaluator Mc, each of the new compiler generators is "partially improved" as compared to a compiler generator CoGen produced from the original metaevaluator Mc. The compiler generator CoGen′ generates a compiler that performs deeper transformation of a program p, but the compiler itself and CoGen′ are of the previous quality. The compiler generator CoGen″ generates a compiler which works with the quality provided by Mc′, but has the quality of the old Mc. Finally, the compiler generator CoGen‴ generates a compiler of the best quality, corresponding to the quality of the new metaevaluator Mc′.

We see two interesting points in the series of MSTs: the third MST (4.2.6) which produces a compiler generator from a given metaevaluator Mc, and the sixth MST (4.3.3) which produces the compiler generator CoGenGGG, which, when applied to Mc′ as in (4.3.5), produces the same compiler generator CoGen‴ that is obtained by the third MST using the new Mc′. Using CoGenGGG (4.3.6) may be more efficient than generating the compiler generator from scratch (4.3.9). However, generating CoGenGGG requires computer resources as well, but may be advantageous if CoGen‴ has to be generated several times (e.g., during the development of a new metaevaluator).

From a Compiler Generator to a Compiler-Generator Generator

In fact, there is no need to generate CoGenGGG by the sixth MST, since the CoGen produced by the third MST can be used for the same purpose. Let us consider this in more detail to illustrate the effect of *degeneration of an ultra-MST* and how it may be avoided. Recall that CoGen is applied as follows:

$$\langle\langle\langle \texttt{CoGen int}\rangle\ \texttt{p}\rangle\ \texttt{x}\rangle \Rightarrow \texttt{y}$$

Indeed, the compiler generator CoGen can be applied not only to interpreters, but to any program of two arguments, e.g. the specializer Spec (Section 3.3). Let the specializer Spec′ be defined through the new metaevaluator Mc′:

$$\textbf{def}\ \langle \texttt{Spec}'\ \texttt{p, x}\rangle = \langle \texttt{Mc}'\ \texttt{p}\ \texttt{x}\ \langle \bullet\ \bullet, \texttt{y}\rangle\rangle$$

The compiler generator CoGen applied to the specializer Spec′, produces the "partially improved" compiler generator, CoGen′, which, when 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‴:

$$\langle \texttt{CoGen Spec}'\rangle = \langle \texttt{Mc}\ \texttt{Spec}'\ \langle \texttt{Mc}\ \texttt{p}\ \langle \bullet\ \bullet, \texttt{x}\rangle\rangle\rangle$$

$$= \langle \texttt{Mc}\ \langle \texttt{Mc}\ \texttt{p}\ \langle \texttt{Mc}'\ \texttt{p}\ \langle \bullet\ \bullet, \texttt{y}\rangle\rangle\rangle = \texttt{CoGen}'$$

$$\langle \texttt{CoGen}'\ \texttt{Spec}'\rangle = \langle \texttt{Mc}\ \langle \texttt{Mc}'\ \texttt{p}\ \langle \texttt{Mc}'\ \texttt{p}\ \langle \bullet\ \bullet, \texttt{y}\rangle\rangle\rangle = \texttt{CoGen}''$$

$$\langle \texttt{CoGen}''\ \texttt{Spec}'\rangle = \langle \texttt{Mc}'\ \langle \texttt{Mc}'\ \texttt{p}\ \langle \texttt{Mc}'\ \texttt{p}\ \langle \bullet\ \bullet, \texttt{y}\rangle\rangle\rangle = \texttt{CoGen}'''$$

So, we see that the compiler generator CoGen‴ can be generated from a new metaevaluator Mc′ by an existing compiler generator CoGen, using an intermediate definition of Spec′ (Klimov and Romanenko, 1987). Thus, we have expressed the fourth and higher MSTs by applying the result of the previous MSTs (a compiler generator) to the metaevaluator in the form of Spec′.

$$\begin{aligned} &\langle \texttt{CoGen Spec}'\rangle && \Rightarrow \texttt{CoGen}' \\ &\langle\langle \texttt{CoGen Spec}'\rangle\ \texttt{Spec}'\rangle && \Rightarrow \texttt{CoGen}'' \\ &\langle\langle\langle \texttt{CoGen Spec}'\rangle\ \texttt{Spec}'\rangle\ \texttt{Spec}'\rangle && \Rightarrow \texttt{CoGen}''' \end{aligned}$$

Degeneration of Ultra-MST

The fourth and the subsequent MSTs above return the same compiler generator as the third MST (4.2.6) if applied to the same metaevaluator Mc: CoGen =

⟨CoGen Spec⟩, and the series of MSTs comes to a fixed point. This is an instance of the general effect of degeneration of an MST series, which usually occurs when an ultra-MST is performed by a formal scheme without adding new information ("creative knowledge") at each step: then, in a finite number of steps, it comes either to a fixed point, or to a state in which the scheme is not applicable. On the other hand, if new information is added, then the MST series does not degenerate. This can be seen from the example of ultra-MST above. If we apply the compiler generator CoGen to a new metaevaluator Mc′, given in the form of a specializer, a compiler generator CoGen′ with new qualities is generated.

MST Over Ultra-MST

If the MST scheme of an ultra-MST becomes the object of metacomputation, then this is an activity which is one MST level higher than the ultra-MST. An example of such an activity can be shown in our case. Indeed, applying the MST scheme to the same metaevaluator Mc′ several times, one can perform three steps before the ultra-MST degenerates. So, we can construct the corresponding MST-formula for a compiler-generator generator CoGenGen, which, when given an arbitrary Mc, produces the fixed point of CoGen in one step. Here, the **where** clause is introduced because the same specializer spec is used three times:

```
def ⟨CoGenGen mc⟩ =
        ⟨⟨⟨CoGen spec⟩ spec⟩ spec⟩ where spec = ⟨mc mc ⟨• p x⟩⟩
                                                  ⟨• •,y⟩
```

As usual, metacomputing this definition generates a compiler-generator generator CoGenGen′ which, given a metaevaluator Mc, immediately produces a new fixed point:

```
⟨Mc ⟨⟨⟨CoGen spec⟩ spec⟩ spec⟩ where spec = ⟨mc mc ⟨• p x⟩⟩⟩
                                              ⟨• •,y⟩
```

Now, one step of the new ultra-MST corresponds to three steps of the original ultra-MST. We have compressed three MSTs into a single one.

5. APPLICATIONS TO MATHEMATICS

This section is devoted to applications of MST-schemes to linguistic modeling in mathematics. Upon considering the problem of manipulating hierarchies of mathematical definitions, which is closely related to the conversion of compilers into interpreters, we show how metacomputation can be used for proving mathematical theorems.

5.1. Reducing Hierarchies of Mathematical Definitions

An illustrative example which requires the conversion of compilative definitions to interpretive ones can be found in mathematics. Definitions provided in mathe-

matical textbooks are *compilative*: they define the formal meaning of a new notion in terms of existing ones. The meaning of a new term is determined by *replacing* it with the terms used in its definition, that is, by compiling. Of course, such translations are not actually performed by mathematicians; instead they use informal images, or mental *interpretation* while working with these extended languages. But, formally, mathematical theories are hierarchies of compilative definitions.

An excellent example is provided by the treatise *Elements of Mathematics* (Bourbaki, 1960), whose first volume, in the first chapter, defines a basic language for the whole of mathematics and the notion of correct proofs. The later is defined through an algorithm that checks whether a given text `t` written in the basic language is a correct proof. We shall call this algorithm `BasicProof`:

$$\langle \texttt{BasicProof t} \rangle \Rightarrow \texttt{True or False}$$

The text of Bourbaki's 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 languages. This sequence of definitions, say `D`, is actually a compiler description which, given as input to an algorithm that expands definitions, say `Expand`, translates a text `t` into a text `t'` in the basic language:

$$\langle \texttt{Expand D, t} \rangle \Rightarrow \texttt{t'}$$

A text `t` is considered to be a correct proof, if the text `t'` obtained by compiling `t` to the basic language is a correct proof:

$$\langle \texttt{BasicProof } \langle \texttt{Expand D, t} \rangle \rangle \Rightarrow \texttt{True or False}$$

So, we have two kinds of language definitions: an interpreter `BasicProof` that assigns the meaning, `True` or `False`, to a text in the basic language, and a compiler `Expand` that translates a text in the extended language defined by `D` to the basic language (terms not defined by `D` are left unchanged).

Why not perform the expansion and the basic proof checking by a computer? An attempt was made in 1969 by V. Turchin and S. Romanenko to expand the first texts of the first chapter of Bourbaki's treatise into the basic language. But the experiment failed! It was found that any realistic computer memory will be immediately exhausted, if not with the texts of the first chapter then with the texts of the following chapters, simply because the size of the expanded text grows exponentially with the height of the hierarchy of definitions. This indicates that it is practically impossible to verify Bourbaki's mathematical treatise directly.

Is there any chance to verify the text by a computer? Mathematicians check the text by interpreting it in their minds without first compiling it to the basic language. But these interpretation rules are not explicated in the treatise. This is not accidental: when a language is gradually extended by adding new notions step by step, it is usually much simpler to define formal rules for translating them into existing notions than to define rules for interpreting a text in the extended language.

Thus, there is a need to derive a new *interpreter* `ProofD` for the extended language, given the interpreter `BasicProof` for the basic language and the compiler `Expand` which is parameterized with respect to the definitions `D`. In principle, a special interpreter could be constructed by hand in order for mathematical texts written in an extended language to be interpreted more efficiently. But, as argued above, the

possibility of introducing new definitions at any time is an essential feature of linguistic modeling which can only be achieved if the manipulation of linguistic processors is fully mechanized.

5.1.1. Generation of Proof-Checkers

The First MST

The interpreter `ProofD` is fully specified by

$$\textbf{def}\ \langle \texttt{ProofD t} \rangle = \langle \texttt{BasicProof} \langle \texttt{Expand D, t} \rangle \rangle \tag{5.1.1}$$

In order to obtain a more efficient interpreter, the definition is metacomputed. If metacomputation is powerful enough, the interpreter `ProofD'` is what is needed:

$$\langle \texttt{Mc} \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\langle \texttt{BasicProof} \langle \texttt{Expand D, t} \rangle \rangle} \rangle \Rightarrow \texttt{ProofD'} \tag{5.1.2}$$

The Second MST

Now let us use the scheme of Section 4 and vary the definition of the compiler. This is actually what Bourbaki's formalization of mathematics requires: after each new definition, the compilation process changes and a new proof-checker is required. We replace `D` with a variable, say `d`, to define a parameterized MST formula and metacompute it:

$$\langle \texttt{Mc} \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\langle \texttt{Mc} \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\langle \texttt{BasicProof} \langle \texttt{Expand} \bullet \texttt{, t} \rangle \rangle}\ \texttt{d}\ \rangle} \rangle \Rightarrow \texttt{ProofGen'} \tag{5.1.3}$$

The result, `ProofGen`, is a generator of interpreters for mathematical texts. It is an algorithm which, given definitions `D`, produces an algorithm, `ProofD`, which checks whether a given mathematical text using the notions defined in `D`, is a correct proof:

$$\langle \texttt{ProofGen D} \rangle \Rightarrow \texttt{ProofD}, \qquad \langle \texttt{ProofD t} \rangle \Rightarrow \texttt{True or False}$$

5.1.2. Extending the Basic Language

Although the basic language used in Bourbaki's treatise does not change, the language is extended after each definition $\texttt{D}_n$ and can be considered as a new basic language for defining the next language extended by definition $\texttt{D}_{n+1}$. Therefore, the corresponding proof-checker $\texttt{Proof}_{n+1}$ can be generated from the previous proof-checker $\texttt{Proof}_n$:

$$\langle \texttt{Mc} \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\langle \texttt{Proof}_n \langle \texttt{Expand D}_{n+1} \texttt{, t} \rangle \rangle} \rangle \Rightarrow \texttt{Proof}_{n+1} \tag{5.1.4}$$

One may expect that the incremental generation of the proof-checkers is faster than producing $\texttt{Proof}_n$ from the basic language interpreter:

$$\langle \texttt{Mc} \underset{\langle \texttt{BasicProof} \langle \texttt{Expand } D_1 \ldots D_n\ D_{n+1}, t \rangle \rangle}{\_\_\_\_\_} \rangle \Rightarrow \texttt{Proof}_{n+1} \qquad (5.1.5)$$

However, this may be either faster, or slower than using the proof-checker generator ProofGen depending on the complexity of the definition list $D_1 \ldots D_n\ D_{n+1}$:

$$\langle \texttt{ProofGen } D_1 \ldots D_n\ D_{n+1} \rangle \Rightarrow \texttt{Proof}_{n+1}$$

By analogy with the formula (5.1.3), the incremental generator of proof-checkers can be constructed, if a mathematical theory defined as $\texttt{Proof}_n$ is extended in different ways:

$$\langle \texttt{Mc} \underset{\langle \texttt{Mc} \underset{\langle \texttt{Proof}_n \langle \texttt{Expand} \bullet, t \rangle \rangle}{\_\_\_\_\_\texttt{d}\_\_\_\_} \rangle}{\_\_\_\_\_} \rangle \Rightarrow \texttt{ProofGen}_n \qquad (5.1.6)$$

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

$$\langle \texttt{ProofGen}_n\ D_{n+1} \rangle \Rightarrow \texttt{Proof}_{n+1}, \qquad \langle \texttt{Proof}_{n+1}\ t \rangle \Rightarrow \texttt{True or False}$$

The Third MST

We can define the process (5.1.6) for an arbitrary proof-checker $\texttt{Proof}_n$ (we lift it up and replace with a variable, p) and perform the following metacomputation:

$$\langle \texttt{Mc} \underset{\langle \texttt{Mc} \underset{\langle \texttt{Mc} \_\vert \underset{\langle \bullet \langle \texttt{Expand} \bullet, t \rangle \rangle}{\_\_\_\_\_\texttt{d}\_\_\_\_} \rangle}{\_\_\_\texttt{p}\_\_\_} \rangle}{\_\_\_\_\_} \rangle \Rightarrow \texttt{ProofGenGen} \qquad (5.1.7)$$

The result is a the generator of incremental generators of proof-checkers, which, applied to a proof-checker $\texttt{Proof}_n$, produces the same incremental generator as that produced by the formula (5.1.6):

$$\langle \texttt{ProofGenGen Proof}_n \rangle \Rightarrow \texttt{ProofGen}_n$$
$$\langle \texttt{ProofGen}_n\ D_{n+1} \rangle \Rightarrow \texttt{Proof}_{n+1}$$
$$\langle \texttt{Proof}_{n+1}\ t \rangle \Rightarrow \texttt{True or False}$$

5.1.3. Incremental Generation of Proof-Checkers

The series of MSTs appears to be deterministic. However, the development was determined by our own choice. Considering the process of incrementally generating a proof-checker (5.1.4), we can vary not just one D_{n+1}, but both $\texttt{Proof}_n$ and D_{n+1} (they are replaced with variables p and d respectively):

$$\langle \texttt{Mc} \underset{\langle \bullet \langle \texttt{Expand} \bullet, t \rangle \rangle}{\_\texttt{p}\_\_\_\_\_\texttt{d}\_\_\_\_} \rangle \Rightarrow \texttt{p}' \qquad (5.1.8)$$

and then, performing the second MST, produce the incremental proof-checker generator:

$$\langle \texttt{Mc} \; \underbrace{\qquad\qquad\qquad\qquad} \rangle \Rightarrow \texttt{ProofIncGen} \qquad (5.1.9)$$
$$\langle \texttt{Mc} \; \underbrace{\texttt{p}\qquad\qquad\texttt{d}} \rangle$$
$$\langle \bullet \; \langle \texttt{Expand} \; \bullet, \texttt{t} \rangle \rangle$$

The result ProofIncGen applied to a proof-checker $\texttt{Proof}_n$ and a new definition D_{n+1}, generates the proof-checker for the extended language, $\texttt{Proof}_{n+1}$:

$$\langle \texttt{ProofIncGen Proof}_n, D_{n+1} \rangle \Rightarrow \texttt{Proof}_{n+1}$$

Conclusion

All MST-formulas considered above are based on the following metacomputation operations: program composition, specialization, and self-application of metacomputation. If the metacomputation methods are powerful enough to reduce hierarchies of mathematical definitions, then they will be useful for solving a variety of other, non-mathematical problems. The main difficulty is the repeated metacomputation of program composition. We also saw that MSTs are not restricted to a single MST series, as different schemes can be defined, provided that one can vary several parts of the initial formula. In the case of definition hierarchies, a series of languages is produced by step-wise extensions. MST-formulas can be used to formalize a variety of other problems, but this is beyond the scope of this paper.

5.2. Theorem Proving by Metacomputation

The axiomatic method of representing mathematical knowledge is not always adequate for manipulation by metacomputation. Instead of defining axioms that state properties of objects and functions, and then giving an algorithm of verifying proofs, mathematical objects can be represented directly as linguistic expressions, and predicates and functions as programs. This is the essence of the *constructive approach* as opposed to the *formal axiomatic* one. Indeed, a certain class of mathematical theorems can be proven directly by metacomputation (Turchin, 1977; 1980b), a goal that has driven the development of supercompilation (Turchin, 1986).

Consider the representation of the natural numbers in the unary number system, where the number n is a string consisting of the symbol 0 followed by n quotes (e.g., 3 is represented as 0'''). Instead of using the Peano axioms for formal arithmetic, the arithmetic operations are defined as algorithms. For example, addition and equality of two natural numbers are defined as (sentences ordered by priority)

```
def ⟨Add x,0⟩  = x                  def ⟨Eq 0,0⟩  = True
    ⟨Add x,y'⟩ = ⟨Add x,y⟩'             ⟨Eq x',y'⟩ = ⟨Eq x,y⟩
                                        ⟨EQ x,y⟩  = False
```

The predicate expressing the commutativity of addition

$$x + y = y + x$$

is defined as a program Comm:

```
def ⟨Comm x,y⟩ = ⟨Eq ⟨Add x,y⟩,⟨Add y,x⟩⟩        (5.2.1)
```

Then, the theorem that addition is commutative can be formulated as the statement that the program `Comm` returns the value `True` for all natural numbers `x` and `y`. As long as the predicate `P` is defined algorithmically, any theorem of the form

$$\forall x,y,\ldots,z\colon P(x,y,\ldots,z)$$

can be expressed in the same way. To prove such a theorem, the predicate `P` is metacomputed

$$\langle \texttt{Mc} \ \frac{\qquad\qquad}{\langle \texttt{P x, y, ..., z} \rangle} \rangle \Rightarrow \texttt{P}' \tag{5.2.2}$$

If the algorithm `P'` obtained by metacomputation has a certain form, from which it is clear that `P'` always returns `True`, then we conclude that the theorem holds. The simplest case is obviously a program of the form

```
def ⟨P x, y, ..., z⟩ = True
```

which immediately returns `True`. If the algorithm `P'` does not fit into the required form, the theorem is not necessarily false, but is merely means that the theorem could not be proven by metacomputation. In other words, the power of the metacomputation method was insufficient to reduce `P`.

Metasystem Structure of the Two Approaches

The axiomatic representation of mathematical knowledge has two parts: a set of axioms (or axiom schemes) and a deductive system. They are organized as a metasystem with the former being at the object level and the latter at the control level. Traditionally, mathematics expresses the majority of its knowledge by axioms at the object level. The deductive system is minimal and consists of a few deductive rules, e.g. the Modus Ponens:

`P, P implies Q`	—two statements which were already proven
`Q`	—the statement which is then proven

In the constructive approach all functions and predicates are defined as algorithms at the object level rather than as axioms. Instead of deduction rules, the control lies in the metaevaluator: the "division of labor" is inverse. Although the definitions at the object level are usually similar to the axioms (see above), the control level is much richer. Moreover, it is not fixed, it evolves. No metaevaluator `Mc` can prove all theorems because this is an algorithmically unsolvable problem, but every new metaevaluator `Mc` can achieve more and more. Besides, a metaevaluator includes some of the mathematical knowledge that is traditionally represented by axioms. The induction principle is a good example to illustrate such knowledge.

Principle of Induction

The principle of induction is usually formulated in form of an axiom scheme. Although the idea behind the induction principle is the same for all theories, the

axiom scheme has to be defined for each theory anew. Consider the induction principle for natural numbers:

> For any predicate P,
> if P(0) and for all numbers n: P(n) implies P(n+1)
> then P(n) for all numbers n.

This definition relies on the particular constructors of the data domain, 0 and n+1. In another theory which uses a different data domain, the antecedent takes another form.

In the axiomatic approach one can not formalize the general principle of induction: if the data domain is built by repeated application of constructors (like n+1) starting from atomic data (like 0), and if a property (like P) holds for all atomic data and is preserved by application of any constructor, then the property holds for all data. As pointed out by Turchin, the principle of induction is a meta-principle, rather than a special axiom scheme that has to be added to each theory.

In the constructive approach the induction principle can be formalized at the metalevel for all theories. As shown above, proving theorems by metacomputation takes two steps:

(1) metacomputation of the predicate which is asserted to be always True;
(2) recognition that the resulting program has such a form that it obviously can return only True.

The power of theorem proving can be increased by improving either the metacomputation, or the recognition. Some improvements are easier to formalize in metacomputation, others in recognition. The induction principle is the latter case.

So, the *principle of induction* is the following rule for recognition: if all terminal points in the resulting program are `True`, then the program always returns `True`. Let `Induction` be the algorithm that performs this check. It returns `True` for a program with terminal nodes being `True`; otherwise the answer is `NotProved`:

$$\langle \texttt{Induction "a program with terminal nodes being True"} \rangle \Rightarrow \texttt{True}$$

Then the process of proving a theorem `P` by induction is defined as

$$\langle \texttt{Induction} \langle \texttt{Mc} \underset{\langle \texttt{P x,y,...,z} \rangle}{\_\_\_\_\_} \rangle \rangle \tag{5.2.3}$$

Example

Consider a special case of proving the commutativity of addition, where $y = 1$:

$$x + 1 = 1 + x \tag{5.2.4}$$

Even a simple supercompiler transforms the definition of the corresponding predicate (where 0' is 1)

$$\textbf{def}\ \langle \texttt{Comm1 x} \rangle = \langle \texttt{Eq} \langle \texttt{Add x, 0'} \rangle, \langle \texttt{Add 0', x} \rangle \rangle \tag{5.2.5}$$

into the form:

```
def ⟨Comm1 0⟩ = True
    ⟨Comm1 x'⟩ = ⟨Comm1 x⟩
```

The latter is a program with one terminal point which is `True`; that is, the program returns `True` for all natural numbers. Thus, the commutativity theorem (5.2.4) is proven.

Second MST

As in the previous sections, the MST-formula (5.3.3) can be metacomputed to make the process of theorem proving more efficient. Assume that the varying part is the definition of the predicate `P`. Again, we express this by lifting `P`, and replacing it with a variable, say `p`:

```
⟨Mc ____________________________⟩ ⇒ IndProver
    ⟨Induction ⟨Mc _p_____⟩⟩
                ⟨• x,y,...,z⟩
```

The result of metacomputation is an *inductive theorem prover* `IndProver`. Given the definition of a predicate, it tries to prove that the predicate is `True` for all arguments.

The MST-formula above requires metacomputation of composition of the `Induction` and `Mc`. The expected advantage of the generated `IndProver` over computing the formula (5.2.3) is that the process of metacomputation over `p` can be aborted as soon as one of the terminal nodes is not `True`.

Conclusion

Metacomputation can be used to define what it means that a statement is proven, as well as to directly prove statements by the computer. The class of provable statements grows while the methods of metacomputation evolve. In our view, the constructive approach to theorem proving is more promising in the long run than the traditional approach based on the search of proofs in deductive systems.

6. CONCLUSION

While large software systems are used to solve problems with less human effort and intervention, computer science not been able to fully cope with the problem of software development. This mirrors the fact that the control of executing models has been achieved, while the control over the creation of linguistic models has not been affected directly by the introduction of the computer.

We analyzed metasystem transitions, which may be observed, or are intentionally organized, in computer science and mathematics. In this paper we were especially interested in different schemes for metasystem transition and metacomputation, and potential applications. We say that the next step in formal linguistic modeling, the next large-scale metasystem transition, is achieved if efficient linguistic models can be created by the computer and it suffices for the human to make initial formal

definitions. The ultimate goal is to achieve the ability for an arbitrary series of metasystem transitions over linguistic models to be just an ordinary, mechanical process, where only high-level decisions are taken by the human.

In cybernetics one says that a purposive system has some knowledge, if it has a model of some part of the world in which the system finds itself. Hence, mastering the method of linguistic modeling means providing new tools for steering and improving the development of knowledge. But it also means advancing the scientific method—with all its consequences.

Dedication

This paper is dedicated to the memory of Alexander Romanenko.

Acknowledgments

This work could not have been carried out without the pioneering work of Valentin Turchin. We are very grateful for many inspirations and stimulating discussions. Sergei Romanenko gave valuable advice on an earlier version of the paper. It is a pleasure to acknowledge useful discussions with the members of the Refal group in Moscow and the Topps group at DIKU. Special thanks are due to Sergei Abramov, Ruten Gurin, Victor Kistlerov, Arkady Klimov, Kristian Nielsen, Tom Reps, and David Sands.Many thanks to Francis Heylighen, Cliff Joslyn, and Valentin Turchin, the editors of the Principia Cybernetica Project.

Robert Glück is supported by the Austrian Science Foundation (FWF) under grant numbers J0780 and J0964. E-mail: glueck@diku.dk.

Andrei Klimov is supported by the Russian Basic Research Foundation under grant number 93-01-628 and in part by the "Österreichische Forschungsgemeinschaft" under grant number 06/1789. E-mail: And.Klimov@refal.msk.su.

References

Abramov, S. M. 1991. Metavychislenija i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3, 31–44 (in Russian).

Bourbaki, N. 1960. Éléments de Mathématique. Théorie des Ensembles. Vol. Premier Partie, Livre I, Hermann.

Futamura, Y. 1971. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5), 45–50.

Futamura, Y. 1983. Partial Computation of Programs. In Goto, E., et al. (eds.): *RIMS Symposia on Software Science and Engineering*. Kyoto, Japan. Lecture Notes in Computer Science, 147, 1–35, Springer-Verlag.

Glück, R. 1991. Towards Multiple Self-Application. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. New Haven, CT: ACM Press, pp. 309–320.

Glück, R. 1992. The Requirement of Identical Variety. In: *Proceedings of the 13th International Congress on Cybernetics*. Namur, Belgium: International Association for Cybernetics, pp. 524–529.

Glück, R. 1994. On the generation of specializers. *Journal of Functional Programming*, 4(4), 499–514.

Glück, R., and Jørgensen, J. (1994). Generating Optimizing Specializers. In: *1994 International Conference on Computer Languages*. Toulouse, France: IEEE Computer Society Press, pp. 183–194.

Glück, R., and Klimov, A. V. 1994. Metacomputation as a Tool for Formal Linguistic Modeling. In Trappl, R. (ed.): *Cybernetics and Systems '94*, Vol. 2. Singapore: World Scientific, pp. 1563–1570.

Harrison, P. G. 1988. Function Inversion. In Bjørner, D., Ershov, A. P., and Jones, N. D. (eds.): *Partial Evaluation and Mixed Computation*. Gammel Avernæs, Denmark: North-Holland, pp. 153–166.

Jones, N. D., Gomard, C. K., Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International Series in Computer Science. New York, London, Toronto: Prentice-Hall.

Jones, N. D., Sestoft, P., and Søndergaard, H. 1985. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In Jouannaud, J.-P. (ed.): *Rewriting Techniques and Applications.* Dijon, France. Notes in Computer Science, 202, pp. 124–140, Springer-Verlag.

Klimov, A. V., and Romanenko, S. A. 1987. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples.) Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow. Preprint No. 71 (in Russian).

Kowalski, R. 1979. Algorithm = logic + control. *Communications of the ACM,* 22(7), 424–436.

Romanenko, A. Y. 1991. Inversion and Metacomputation. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* New Haven, CT: ACM Press, pp. 12–22.

Turchin, V. F. 1972. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal.) In: *Teorija Jazykov i Metody Programmirovanija* (*Proceedings of the Symposium on the Theory of Languages and Programming Methods.*) Kiev-Alushta, USSR, pp. 31–42 (in Russian).

Turchin, V. F. 1977. *The Phenomenon of Science.* New York: Columbia University Press.

Turchin, V. F. 1980a. The language Refal, the theory of compilation and metasystem analysis. Courant Institute of Mathematical Sciences, New York University. Courant Computer Science Report No. 20.

Turchin, V. F. 1980b. The Use of Metasystem Transition in Theorem Proving and Program Optimization. In de Bakker, J. W., and van Leeuwen, J. (eds.): *Automata, Languages and Programming.* Noordwijkerhout, Netherlands. Lecture Notes in Computer Science, Vol. 85, pp. 645–657, Springer-Verlag.

Turchin, V. F. 1986. The concept of a supercompiler. *ACM TOPLAS,* 8(3), 292–325.

Turchin, V. F. 1993. Program transformation with metasystem transitions. *Journal of Functional Programming,* 3(3), 283–313.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering,* 10(4), 352–357.