REDUCTION OF LANGUAGE HIERARCHIES BY METACOMPUTATION

1. INTRODUCTION

One of the defining features of modern science is the use of languages, both informal and formal, to construct *linguistic models* of reality [11]. The introduction of the computer was a revolutionary step in the execution of formal linguistic models and, as a result, the number of linguistic models created and used has significantly increased in all branches of science during the last decades. Computer science, as we see it, is laying the foundations and developing the research paradigm and scientific method of formal linguistic modeling. Linguistic models that can be performed by a computer, at least in principle, are referred to as programs, or algorithms.

Languages and their definitions play a central role in all forms of linguistic modeling. Already in the late fifties the fast growing number of languages and their increasing complexity was felt as "one of the fundamental problems facing the computer profession" [10]. A modern approach in computer science for solving wide-spectrum problems is to devise application-oriented languages that make it easy for the user to express computational tasks in a particular area. Such languages are often defined by using several interpreters, or by translating them to a ground language via a series of intermediate languages. But hierarchies of languages are not a simplification in terms of the underlying designation process, but increase computational complexity: a statement of a higher level language is usually defined by a sequence of actions (or phrases) on a lower, more elementary level. Effective and efficient reduction of language hierarchies is a prerequisite for formal linguistic modeling on a large scale.

This contribution addresses the problem of systematically reducing the computational costs of language hierarchies by two forms of metacomputation: *program composition* and *program specialization*. We shall not be concerned *how*, but *what* has to be achieved by metacomputation. The presented problems can be seen as test cases for existing methods and as a guideline for future work. More specifically, we study the reduction of *homogeneous hierarchies*, *i.e.*, hierarchies of translators or interpreters

only, as well as *heterogeneous hierarchies* in which translative and interpretive definitions may occur in any order. For the sake of completeness, we also summarize two approaches for converting translators into interpreters and vice versa. From now on we shall refer to formal languages simply as languages.

This work belongs to a line of research which aims at a better understanding of metacomputation and the use of metasystem transition, *e.g.*, [4, 5, 6, 7, 11, 12, 13, 14]. In the remainder of this paper we adopt the language-independent formalization of [5] based upon [12, 4, 7].

2. HIERARCHICAL SYSTEMS OF LANGUAGES

Data, programs and application We assume a fixed set of data **D** for input and output, and to represent programs written in different languages. We shall assume nothing further about data; we could choose symbol strings, Lisp lists *etc.* To express the *application* of an L-program $Pgm \in \mathbf{D}$ to its input $Input \in \mathbf{D}$ we use angular brackets $\langle ... \rangle$ and denote computation by writing $\langle Pgm\ Input \rangle_{\mathrm{L}} \Rightarrow Output$. We omit the language index L when it is not essential. Capitalized names denote arbitrary elements of the data domain, *e.g.*, $Pgm \in \mathbf{D}$. Two expressions in the application language are computationally equivalent, $a = b$, if they can be reduced to identical elements of **D** (or both expressions are undefined).

Language definitions We know of exactly two forms of language definitions: interpretive and translative. An interpreter defines a source language A by actions in another language B, while a translator defines a source language A by translation to a target language B where the translation is described in a meta-language M.

Definition 1 (Interpretation). A B-program *IntAB* is an A/B-*interpreter* if for every A-program $P \in \mathbf{D}$ and every input $X \in \mathbf{D}$

$$\langle P\ X \rangle_{\mathrm{A}} = \langle IntAB\ P,\ X \rangle_{\mathrm{B}}$$

Definition 2 (Translation). An M-program *TransAB* is an A→B-*translator* if

1. $\langle TransAB\ P \rangle_{\mathrm{M}} \in$ B-programs for all A-programs $P \in \mathbf{D}$, and
2. $\langle \langle TransAB\ P \rangle_{\mathrm{M}}\ X \rangle_{\mathrm{B}} = \langle P\ X \rangle_{\mathrm{A}}$ for all A-programs $P \in \mathbf{D}$ and every input $X \in \mathbf{D}$.

The following shorthand notation is sometimes used to denote an A→B-translator written in M: $\mathrm{A} \underset{\mathrm{M}}{\rightarrow} \mathrm{B}$.

Language hierarchies A hierarchical system of languages is a series of consecutive definitions where each definition is either an interpreter, or a translator. The hierarchy starts with a language N on the highest level and ends at a ground language 0. This is illustrated below where Def_{I} is either an I/J-interpreter or an I→J-translator (written in some lower language).

We require that each language hierarchy is *closed* in the sense that the meta-languages of the translators can always be reduced to the ground language. For the sake of simplicity, let all translators in a hierarchy be written in the same language M which is identical to the ground language 0. Otherwise, every meta-language can be reduced to the ground language 0 using the methods shown later in this paper.

$$\mathrm{N} \xrightarrow{Def_{\mathrm{N}}} \mathrm{N{-}1} \quad \ldots \quad \mathrm{I} \xrightarrow{Def_{\mathrm{I}}} \mathrm{J} \quad \ldots \quad 1 \xrightarrow{Def_{1}} 0$$

Correct hierarchies Language hierarchies, being sequences of interpreters and translators, obey certain typing rules. Let us denote by ⋆ a *join* operation for building a language hierarchy. For example, $Def_{\mathrm{A}} \star Def_{\mathrm{B}} \star Def_{\mathrm{C}}$ is a three-level hierarchy constructed from three language definitions, where A is defined by B and B is defined by C. The following rules hold for any pair of adjacent definitions in a correct hierarchy, where _/_ is an interpreter, _→_ a translator, and A, B, C are languages. In case (1) and (2) we speak of a *homogeneous* hierarchy, and in case (3) and (4) of a *heterogeneous* hierarchy.

(1)	A→B ⋆ B→C	(3)	A/B ⋆ B→C
(2)	A/B ⋆ B/C	(4)	A→B ⋆ B/C

Complexity of hierarchies Abstraction by means of language hierarchies is not a simplification in terms of time and space complexity. On the contrary, a statement of a higher level language is usually defined by a sequence of actions (or phrases) on a lower, more elementary level. Generally speaking, the computational costs grow exponentially with the height of the interpreter hierarchy, and the size of a translated text grows exponentially with the height of the translator hierarchy. For instance, efficiency problems in interpreter hierarchies are due to the multiple interpretive overhead, each interpretive level multiplying computation time.

The ultimate goal is to minimize the complexity involved in language hierarchies. A hierarchy of formal language definitions is just a 'complicated program' composed from a series of interpreters and translators. Hence, methods of program transformation and optimization can be used to reduce its complexity.

3. METACOMPUTATION

We refer to any process of simulating, analyzing and transforming programs by means of programs as *metacomputation*, a term that underlines the fact that this activity is one level higher than ordinary computation. Programs that carry out these tasks are *metaprograms*. We shall use the term metacomputation in a general way, independent of a specific transformation paradigm.

Abstraction To represent application expressions without specifying all data elements, we introduce *variables*. We refer to an application expression possibly including variables as *configuration*. A variable ranges over the whole domain **D**. We use lower case names to denote variables, *e.g.*, x, y, p. A configuration represents the set of all application expressions obtained by replacing variables by elements of **D**.

Encoding In order to manipulate configurations by means of programs, we define an injective mapping, called *metacoding*, from configurations into the data domain **D** (this is necessary because application symbols and variables are not elements of **D**). We write a horizontal line above an expression to denote its metacode. For example, $\overline{\langle P\ X,\ y\rangle}$ is a metacoded configuration where $P, X \in \mathbf{D}$, and y is a variable. Metacoding in metacomputation corresponds to a Gödel numeration in logics where statements about a theory are encoded in the theory itself.

Metacomputation Let $Meta \in \mathbf{D}$ be some metaprogram. It follows from our notation that $\langle Meta\ \overline{\langle P\ X,\ y\rangle}\rangle \Rightarrow Output$ denotes metacomputation on a metacoded configuration by a metaprogram *Meta*. We say, a variable is *bounded* by metacoding, and it behaves under metacomputation as a universally quantified variable. For better readability, we move metacoded expression one line down for each metacoding:

$$\langle Meta \underline{\qquad\qquad} \rangle \Rightarrow Output$$
$$\overline{\langle P\ X,\ y\rangle}$$

This characterization of metacomputation states nothing about its concrete nature, except that it involves a metaprogram that operates on a metacoded expression. In this paper, we use two cases of metacomputation to reduce language hierarchies, namely *program composition* and *program specialization* (see, *e.g.*, [12, 15, 9, 2]). We will not fix a particular technique for metacomputation, but specify these two metacomputation tasks equationally.

Definition 3 (Program composition). An M-program *Comp* is an A→B-*composer* if for all A-programs $P, Q \in \mathbf{D}$, every input $X, Y \in \mathbf{D}$:

$$\langle Comp \underline{\qquad\qquad\qquad} \rangle_{\mathrm{M}} \Rightarrow R$$
$$\overline{\langle P\ \langle Q\ x\rangle_{\mathrm{A}},\ y\rangle_{\mathrm{A}}}$$

such that

$$\langle R\ X, Y\rangle_{\mathrm{B}} = \langle P\ \langle Q\ X\rangle_{\mathrm{A}}, Y\rangle_{\mathrm{A}}$$

Definition 4 (Program specialization). An M-program *Spec* is an A→B-*specializer* if for every A-program $P \in \mathbf{D}$, every input X, Y:

$$\langle Spec\ \underbrace{\qquad\qquad}_{\langle P\ X, y\rangle_{\mathrm{A}}}\rangle_{\mathrm{M}} \Rightarrow R \quad \text{such that} \quad \langle R\ Y\rangle_{\mathrm{B}} = \langle P\ X, Y\rangle_{\mathrm{A}}$$

Note that these definitions say nothing about the quality of the metacomputation process, but we expect that the transformations performed are more than trivial program composition or trivial program specialization since the success of reducing the complexity of language hierarchies depends on the power of the metacomputation methods. The formulas used here are collectively referred to as *MST-formulas* (MST=metasystem transition [11]).

4. CONVERTING DEFINITIONS TO THEIR DUAL FORM

Language definitions in mathematics usually take the translative form stated in some meta-language (an excellent example is [1]), while language definitions in computer science are often interpretive. Mathematics is an example of a linguistic activity where it is more convenient to define language extensions stepwise by translation into more elementary phrases than by defining the whole extended language interpretively. On the other hand, when defining a new language from scratch, as is often the case in computer science, it is usually easier to give its definition by an interpreter rather than by a translator. However, in both situations the need may arise for converting translators into interpreters, and vice versa. We summarize two approaches for converting language definitions into their dual form.

Converting translators to interpreters [7], [8] Let *TransAB* be an A→B-translator written in C, and let B be defined by a B/C-interpreter *IntBC*. This is a two-level language hierarchy of the form *TransAB* ⋆ *IntBC*. The A→B-translator can be converted into an A/C-interpreter by program composition as follows.

First, define an inefficient A/C-interpreter *IntAC* that performs A-programs in two stages: by translating an A-program p into B and then by interpreting the B-program with input x.

$$\textbf{def}\ \ \langle IntAC\ p, x\rangle_{\mathrm{C}} = \langle IntBC\ \langle TransAB\ p\rangle_{\mathrm{C}}, x\rangle_{\mathrm{C}}$$

Second, the intermediate language B is removed by applying the C→C-composer to this definition, and a more efficient A/C-interpreter $IntAC'$ is thus obtained:

$$\langle Comp \underbrace{\qquad\qquad\qquad\qquad}_{\langle IntBC\ \langle TransAB\ p\rangle_{\mathrm{C}},\ x\rangle_{\mathrm{C}}} \rangle \Rightarrow IntAC'$$

Converting interpreters to translators [3] Let $IntBC$ be an B/C-interpreter and let $Spec$ be a C→C-specializer written in C. The B/C-interpreter can be converted into an B→C-translator by program specialization as follows.

Let P be a B-program, and define a C-program R such that $\langle R\ X\rangle_{\mathrm{C}} = \langle P\ X\rangle_{\mathrm{B}}$ for all $X \in \mathbf{D}$:

$$\textbf{def}\ \ \langle R\ x\rangle_{\mathrm{C}} = \langle IntBC\ P,\ x\rangle_{\mathrm{C}}$$

First MST. An efficient C-program R' may be obtained by specializing the definition of R:

$$\langle Spec \underbrace{\qquad\qquad}_{\langle IntBC\ P,\ x\rangle_{\mathrm{C}}} \rangle \Rightarrow R'$$

Second MST. An B→C-translator $TransBC$, such that $\langle\langle TransBC\ P\rangle\ X\rangle_{\mathrm{C}} = \langle R'\ X\rangle_{\mathrm{C}}$, can be defined by replacing the metacoded program P by a free variable p in the first MST-formula:

$$\textbf{def}\ \ \langle TransBC\ p\rangle_{\mathrm{C}} = \langle Spec \underbrace{\qquad p \qquad}_{\langle IntBC\ \bullet,\ x\rangle_{\mathrm{C}}} \rangle_{\mathrm{C}}$$

The distance between the variable and position • defines now how many times the value of the variable has to be metacoded upon substitution (here p's value needs to be metacoded once in order to obtain $\bar{P}$). An efficient B→C-translator $TransBC'$ may be obtained by specializing the definition of $TransBC$:

$$\langle Spec' \underbrace{\qquad\qquad\qquad\qquad}_{\langle Spec \underbrace{\qquad p \qquad}_{\langle IntBC\ \bullet,\ x\rangle_{\mathrm{C}}} \rangle_{\mathrm{C}}} \rangle \Rightarrow TransBC'$$

5. REDUCING LANGUAGE HIERARCHIES

This section addresses the problem of reducing homogeneous and heterogeneous language hierarchies. We will see that two metacomputation tasks, namely program specialization and program composition, are sufficient to reduce all forms of language hierarchies.

5.1. *Homogeneous Hierarchies of Languages*

We consider homogeneous language hierarchies and state their reduction for the two-level case. The same methods can be used for homogeneous hierarchies of arbitrary height. Recall that in both cases the translator and interpreter resulting from the reduction of the hierarchy can be converted to its dual definition as explained in Section 4.

Translator hierarchies Let each language in a two-level hierarchy be defined by a translator. A text in the top language A is translated by the translator *TransAB* into a text in the lower language B which is translated by *TransBC* into a text written in the ground language C. The language hierarchy has the form $TransAB \star TransBC$. The goal of metacomputation is to remove the construction of intermediate phrases between the translators. For simplicity, we assume that both translators are defined in the same meta-language M.

An A→C-translator *TransAC* is defined by application of the two translators to an A-program p:

$$\textbf{def}\ \ \langle TransAC\ p\rangle_{\mathrm{M}} = \langle TransBC\ \langle TransAB\ p\rangle_{\mathrm{M}}\rangle_{\mathrm{M}}$$

A more efficient A→C-translator $TransAC'$ that translates A directly into C without the intermediate translation into B can be derived by metacomputation of this program composition:

$$\langle Comp\underline{\qquad\qquad\qquad\qquad\qquad\qquad}\rangle \Rightarrow TransAC'$$
$$\langle TransBC\ \langle TransAB\ p\rangle_{\mathrm{M}}\rangle_{\mathrm{M}}$$

Interpreter hierarchies Let each language in a two-level hierarchy be defined by an interpreter. A text in the top language A is interpreted by the interpreter *IntAB* written in the lower language B which in turn is interpreted by the interpreter *IntBC* described in the language C. The language hierarchy has the form $IntAB \star IntBC$. The goal of metacomputation is to remove intermediate actions between the interpreters.

An A/C-interpreter, which applies an A-program p to its input x, is defined as follows:

$$\textbf{def}\ \ \langle IntAC\ p,\ x\rangle_{\mathrm{M}} = \langle IntBC\ IntAB,\ (p,\ x)\rangle_{\mathrm{C}}$$

A more efficient A/C-interpreter $IntAC'$ that interprets A directly in C without the intermediate interpretation of B can be obtained by specialization of the interpreter *IntBC* with respect to *IntAB*:

$$\langle Spec\underline{\qquad\qquad\qquad\qquad\qquad\qquad}\rangle \Rightarrow IntAC'$$
$$\langle IntBC\ IntAB,\ (p,\ x)\rangle_{\mathrm{C}}$$

5.2. *Heterogeneous Hierarchies of Languages*

We now consider the general case of heterogeneous hierarchies that consist of translative and interpretive definitions in an arbitrary order and show that all heterogeneous hierarchies can be reduced in three steps either to an interpreter or a translator. A heterogeneous hierarchy contains at least an interpreter and a translator.

(1) Interpreter over translators First, each subhierarchy of the form 'interpreter over translator' is reduced to a single interpreter that replaces both definitions. Assume that the language A is interpreted by an A/B-interpreter *IntAB* which in turn is defined by a B→C-translator. Thus, the language hierarchy has the form $IntAB \star TransBC$. An A/C-interpreter *IntAC* which interprets A directly in C can be obtained by ordinary computation: by translating the interpreter *IntAB* into C.

$$\langle TransBC\ IntAB \rangle = IntAC$$

By repeatedly using this reduction, any subhierarchy of the form

$$\ldots \underbrace{IntI\_ \star Trans\_\_ \star \ldots \star Trans\_J}_{\text{interpreter over translators}} \ldots$$

where **I** is the top language and J the ground language, can be reduced to a single **I**/J-interpreter.

(2) Translators over interpreters It is easy to verify that the hierarchy resulting from exhaustively applying step (1) consists of exactly two homogeneous subhierarchies: a sequence of translators followed by a sequence of interpreters. In other words, all translators 'under' interpreters disappeared.

$$\underbrace{TransN\_ \star \ldots \star Trans\_I}_{\text{translative part}} \star \underbrace{IntI\_ \star \ldots \star Int\_0}_{\text{interpretive part}}$$

Further reduction requires metacomputation over each subhierarchy. The two homogeneous subhierarchies can be reduced to a translator (by a program composer) and to an interpreter (by a program specializer), respectively, using the methods explained in Section 5.1.

(3) Two-level hierarchy After step (2) the language hierarchy is reduced to a two-level hierarchy consisting of one translator *TransNI* and one interpreter *IntI0*.

$$\underbrace{TransNI \star IntI0}_{\text{two levels}}$$

The final reduction to a single interpreter can be obtained using a 0→0-composer *Comp* as shown in Section 4 (recall that we assumed that the meta-language of translators can be reduced to the ground language 0):

$$\langle Comp \frac{}{\langle IntI0\ \langle TransNI\ p\rangle_0,\ x\rangle_0}\rangle \Rightarrow IntN0'$$

As a result, the heterogeneous hierarchy consisting of translative and interpretive definitions in an arbitrary order has been reduced to a single interpreter. If necessary, the interpreter *IntN0'* can be converted to an N→0-translator *TransN0* (Section 4).

6. SUMMARY

Table 1 summarizes the reduction formulas stated in the previous sections. In each case a pair of interpreters/translators is transformed into a single definition.

Formulas 1 and 2 describe the two homogeneous cases, while Formulas 3 and 4 describe the two heterogeneous cases. Formula 5 describes the conversion of an interpreter into a translator. The conversion of a language definition into its dual form is covered by Formulas 4 and 5. Note that no direct conversion of a translator into an interpreter exists; an additional interpreter for the target language must be provided in Formula 4. Only one reduction (Formula 3) can be achieved by ordinary computation. All other cases require metacomputation.

We list only the simple cases in the table: a translator is produced in Formula 1 and an interpreter in Formulas 2–4. If the dual form of a definition is needed, it can be obtained by the second step: the $\mathrm{A} \underset{\mathrm{M}}{\rightarrow} \mathrm{C}$-translator produced in Formula 1 can be converted into an A/M-interpreter given a B/M-interpreter according to Formula 4, and the interpreters produced in Formulas 2–4 can be converted into the corresponding translators according to Formula 5.

The formulas presented in Table 1 can be seen as test cases for existing metacomputation methods and as a guideline for further research: they state *what* has to be achieved by metacomputation. We see that a program composer must be capable of

- fusing two translators for non-trivial languages into a single translator (Formula 1),
- fusing a translator and an interpreter into an interpreter (Formula 4).

A program specializer must be capable of

- fusing two interpreters (Formula 2)

– converting an interpreter into a translator (Formula 5).

Practical progress has been achieved with the problems stated in Formula 2 and 5 (*cf.* [9]), while more powerful method program composers are still under development (*cf.* [12, 15, 2]).

TABLE I
Summary of reduction formulas for language hierarchies

	Type of reduction	*MST-formula*	*Note*
1	$A \underset{M}{\to} B * B \underset{M}{\to} C \Rightarrow A \underset{M}{\to} C$	$\langle Comp \overline{\langle TransBC\ \langle TransAB\ p\rangle_M\rangle_M}\rangle \Rightarrow TransAC$	
2	$A/B * B/C \Rightarrow A/C$	$\langle Spec \overline{\langle IntBC\ IntAB,\ (p, x)\rangle_C}\rangle \Rightarrow IntAC$	
3	$A/B * B \underset{M}{\to} C \Rightarrow A/C$	$\langle TransBC\ IntAB\rangle \Rightarrow IntAC$	ordinary computation
4	$A \underset{M}{\to} B * B/M \Rightarrow A/M$	$\langle Comp \overline{\langle IntBM\ \langle TransAB\ p\rangle_M,\ x\rangle_M}\rangle \Rightarrow IntAM$	translator to interpreter
5	$B/C \Rightarrow B \underset{M}{\to} C$	$\langle Spec' \overline{\langle Spec \overline{\langle IntBC\ \bullet,\ x\rangle_C}^{\,p}\rangle_C}\rangle \Rightarrow TransBC$	interpreter to translator

7. CONCLUSION

We studied the structure of language hierarchies and showed that two metacomputation tasks, namely program specialization and program composition, are sufficient to reduce all language hierarchies constructed from interpreters and translators. We argued that a practical solution to this problem is a prerequisite for formal linguistic modeling on a large scale.

It is instructive to compare today's view about programming languages with the dream of the sixties—an aim sometimes still pursued—that *one universal language* would be sufficient to satisfy all programming needs. In the meantime, it became clear that new languages emerge as new problems are attacked. A language suited for one purpose is not necessarily the best for another problem. One can never be sure of finding what is needed to solve a problem among those formalisms that have already been created. The creation of a *variety of languages* is the very nature of linguistic modeling.

A substantial amount of work in computer science has been devoted to the development of high-level formalisms, while searching for ways to produce efficient implementations. Some progress has been achieved, but the needs are still far beyond the actual achievements.

The approach advocated in this paper is the development of a set of *universal tools* for metacomputation, instead of singular solutions for particular (classes of) languages. Metacomputation tools are needed to deal with languages as material, to manipulate languages definitions, and to reduce the complexity of language hierarchies. The problem is *how* to deal with a large variety of languages, and *not* how to find one universal ground language. Indeed, what is needed most are powerful *metaprogramming languages* that allows the construction of metacomputation tools. As explained in [7], three main metacomputation tasks must be performed efficiently: program composition, program specialization, and program inversion.

To what extent the metacomputation approach will be realized, can only be seen in the future. But one thing is for sure, if we are not able to reduce the complexity and costs of language hierarchies, we will never be able to master linguistic modeling to its full extent and achieve the next metasystem transition: mastering the creation of linguistic models on a large scale.

ACKNOWLEDGEMENTS

Partly supported by the project "Design, Analysis and Reasoning about Tools" funded by the Danish Natural Sciences Research Council. The first author was also supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foudation (FWF) grant J0780 & J0964; the second author by the Russian Basic Research Foundation grant 93-01-00628 & 96-01-01315, and the US Civilian Research and Development Foundation grant RM1-254.

Robert Glück
DIKU, Departement of Computer Science,
University of Copenhagen, Universitetsparken 1,
DK-2100 Copenhagen, Denmark
email: glueck@diku.dk

Andrei Klimov
Keldysh Institute of Applied Mathematics,
Russian Academy of Sciences, Miusskaya Square 4,
RU-125047 Moscow, Russia
email: klimov@keldysh.ru

REFERENCES

[1] **Bourbaki, N.**, *Éléments de Mathématique. Théorie des Ensembles*, Premier Partie, Livre I. Hermann, 1960.

[2] **Fegaras, L., Sheard, T. & Zhou, T.**, "Improving programs which recurse over multiple inductive structures." In: *ACM SIGPLAN Work-*

shop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, 1994, pp. 21–32.

[3] **Futamura, Y.**, "Partial evaluation of computing process—an approach to a compiler-compiler" *Systems, Computers, Controls*, **2** (5), 1971, pp. 45–50.

[4] **Glück, R.**, "Towards multiple self-application." In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, 1991, pp. 309–320.

[5] **Glück, R.**, "On the mechanics of metasystem hierarchies in program transformation." In: **Proietti, M.** (ed.), *Logic Program Synthesis and Transformation (LoPSTr'95)*, Lecture Notes in Computer Science, vol. **1048**, Springer-Verlag, 1996, pp. 234-251.

[6] **Glück, R. & Klimov, A.V.**, "Occam's razor in metacomputation: the notion of a perfect process tree." In: **Cousot, P., Falaschi, M., Filè, G. & Rauzy, G.** (eds), *Static Analysis. Proceedings. Lecture Notes in Computer Science*, **724**, Springer-Verlag, 1993, pp. 112–123.

[7] **Glück, R. & Klimov, A.V.**, "Metasystem transition schemes in computer science and mathematics", *World Futures: the Journal of General Evolution*, **45**, 1995, pp. 213-243.

[8] **Glück, R. & Klimov, A.V.**, 'On the degeneration of program generators by program composition", *New generation computing*, **16**(1), 1998, pp. 75-95.

[9] **Jones, N.D., Gomard, C.K. & Sestoft, P.**, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

[10] **Strong, J. Wegstein, J., Tritter, A., Olsztyn, J., Mock, O. & Steel, T.**, "The problem of programming communication with changing machines: a proposed solution" *Communications of the ACM*, **1** (8, 9), 1958, pp. 12-18, pp. 9-15.

[11] **Turchin, V.F.**, *The Phenomenon of Science*, Columbia U. Press, NY, 1977.

[12] **Turchin, V.F.**, "The concept of a supercompiler", *Transactions on Programming Languages and Systems*, **8** (3), 1986, pp. 292-325.

[13] **Turchin, V.F.**, "A constructive interpretation of the full set theory", *The Journal of Symbolic Logic*, **52** (1),1987, pp. 172-201.

[14] **Turchin, V.F.**, "On cybernetic epistemology", *Systems Research*, **10** (1), 1993, pp. 3-28.

[15] **Wadler, P.**, "Deforestation: transforming programs to eliminate trees", *Theoretical Computer Science*, **73**, 1990, pp. 231-248.