New Generation Computing, 16 (1998) 075-096
OHMSHA, LTD. and Springer-Verlag

On the Degeneration of Program Generators by Program Composition*

Robert GLÜCK
Department of Computer Science,
University of Copenhagen,
Universitets parken 1
DK-2100 Copenhagen, Denmark.
Andrei KLIMOV
Keldysh Institute of Applied Mathematics,
Russian Academy of Sciences,
Miusskaya Square 4
RU-125047 Moscow, Russia.

Received 8 July 1996
Revised manuscript received 27 February 1997

Abstract One of the main discoveries in the seventies was that the concept of a generating extension covers a very wide class of apparently different program generators. Program specialization, or partial evaluation, is powerful because it provides uniform techniques for the automatic implementation of generating extensions from ordinary programs. The Futamura projections stand as the cornerstone of the development of program specialization.

This paper takes the idea of the Futamura projections further. Three *degeneration projections* are formulated which tell us how to achieve the reverse goal by program composition, namely turning a generating extension into an ordinary program. The fact that program composition can invert the effect of program specialization shows that these projections are dual in a sense. The degeneration projections complete a missing link between programs and generating extensions and allow for novel applications of program transformation.

* This work was 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 Foundation (FWF) under grant J0780 & J0964; the second author partly by the Russian Basic Research Foundation grant 93-01-00628 & 96-01-01315, the US Office of Naval Research grant N00014-96-1-0800, and the US Civilian Research and Development Foundation grant RM1-254.

Keywords: Programming Languages, Program Transformation, Partial Evaluation, Program Specialization, Program Composition, Metacomputation.

§1 Introduction

One of the main discoveries in the seventies was that the concept of a *generating extension*[1] covers a very wide class of apparently different program generators. This has the big advantage that program generators for diverse applications such as parsing, translation, theorem proving, and pattern matching can be implemented with uniform techniques.[2] *Program specialization*, or partial evaluation, is powerful because it provides transformation techniques for the automatic implementation of generating extensions.[3,4,5] Program specialization can now be considered as one of the most advanced techniques for automatic program transformation. The *Futamura projections*[6] stand as the cornerstone of this development.

This paper takes the idea of the Futamura projections further. Three *degeneration projections* are formulated which tell us how to reverse the effect of the Futamura projections by *program composition*, namely how to turn a generating extension into an ordinary program. The degeneration projections, similar in structure to the Futamura projections but their inverse, complete a missing link between programs and generating extensions. This is interesting in its own right and allows for several novel applications of program specialization and program composition. It is quite remarkable that, although program specialization has been used for some time to generate generating extensions, the reverse operation, degeneration, has not been studied before.

In this paper we shall mainly be concerned with *what* can be achieved by program composition, and not *how* it can be achieved. Although promising results have been obtained for program composition,[7,8,9,10] this method is still at a research stage. The transformation problems presented here can be seen as test cases for existing methods, and as challenging goals for future research.

Suppose P is a program with two arguments and let $\langle P\ X,\ Y\rangle$ denote the application of program P to its input X, Y. Computation of P producing result *Out* is described by

$$\langle P\ X,\ Y\rangle \Rightarrow Out$$

A *generating extension of* P is a program Gen_P that takes one part of P's input, say X, and produces another program P_X, the specialization of P to that input. The specialized program returns the same result when applied to the remaining input Y as the original program P when applied to input X, Y. Computation in two stages is described by

$$\langle Gen_P\ X\rangle \Rightarrow P_X$$
$$\langle P_X\ Y\rangle \Rightarrow Out$$

$\langle P\ X,\ Y\rangle$	$\langle\langle Gen_P\ X\rangle\ Y\rangle$
$\langle$*Parser Grammar, Phrase*$\rangle$	$\langle\langle$*Parsergen Grammar*$\rangle$ *Phrase*$\rangle$
$\langle$*Prover Axioms, Theorem*$\rangle$	$\langle\langle$*Provergen Axioms*$\rangle$ *Theorem*$\rangle$
$\langle$*Matcher Pattern, String*$\rangle$	$\langle\langle$*Matchergen Pattern*$\rangle$ *String*$\rangle$
$\langle$*Interpreter Program, Data*$\rangle$	$\langle\langle$*Translator Program*$\rangle$ *Data*$\rangle$

Fig. 1 Programs and their generating extensions.

Combining these two we obtain the equation

$$\underbrace{\langle P\ X,\ Y\rangle}_{\textit{one stage}} = \underbrace{\langle\langle Gen_P\ X\rangle\ Y\rangle}_{\textit{two stages}}$$

Many computational problems can be solved either by a one stage computation, or by a two-stage computation. Some applications are illustrated in Fig. 1. A two-stage computation can be advantageous if Y changes more frequently than X. The second stage can often be optimized based on the input available in the first stage.

The division of a computation into two (or more) stages has been studied intensively in the area of partial evaluation.[3,4,5)] Futamura[6)] was the first who saw the possibility to obtain generating extensions from general programs by self-application of a program specializer. After a period of independent insights,[6,11,1)] the last decade has seen substantial progress both in theory and practice of program specialization. Automatic tools for turning programs into generating extensions now exist for various programming languages such as Prolog, Scheme, and C (see Ref. 4)). This paper considers the reverse transformation using program composition.

Function composition and specialization (by fixing parameters) is classical mathematics. However, mathematics treats these operations extensionally, paying no attention to the structure of definitions. On the other hand, program composers and program specializers are *software tools* which modify the structure of programs with the purpose of optimizing some aspects of the programs performance, e.g. time and space efficiency.

The remainder of this paper is organized as follows. Section 2 defines the basic notions, Section 3 presents metacomputation, and Section 4 defines program specialization and program composition. Section 5 reviews the generation of generating extensions by program specialization, and Section 6 studies degeneration by program composition. Sections 7 and 8 discuss the corresponding tools. Sections 9 and 10 investigate applications and related aspects. Section 11 gives certain optimality criteria and Section 12 summarizes the projections for generation and degeneration.

§2 Basic Notions

We now formulate the properties of generating extensions, program specializers, and program composers more precisely. A precise notation is necessary since more than one program may be involved at the same time and programs play multiple roles: as active agents and as passive data.

Early notations for describing the interaction of interpreters and translators include the T-diagram.[12,13] In the area of partial evaluation, the Futamura projections are often described using the mix-notation,[4] a formalism inspired by both recursive function theory and Lisp. However, there are several other ways of combining program transformers, such as the degeneration problem considered in this paper, for which the traditional notations are not sufficient.[15] This paper adopts the language-independent notions[14,15] based upon Turchin's MST-schemes.[16]

2.1 Data, Programs and Application

We assume a fixed set of *data* for input and output, and for representing programs written in different languages. We shall assume nothing further about data; we could choose symbol strings, Lisp lists etc.

To express syntactically the *application* of an L-program to its input we use angular brackets $\langle ... \rangle_L$ (we omit the language index L when it is not essential). Computation (reduction) of an application expression is denoted by $\Rightarrow$. For example, $\langle P\ X,\ Y\rangle_L \Rightarrow Out$ is the reduction of program P with input X, Y to output Out. Capitalized names, e.g. P, X, Y, stand for literal constants which represent arbitrary elements of the data domain.

Two application expressions $\mathcal{A}$, $\mathcal{B}$ are *computationally equivalent*, $\mathcal{A} = \mathcal{B}$, if they can be reduced to identical data elements, or both sides are undefined.

Definition 1 (interpreter)
A B-program *Int* is an A/B-interpreter if for every A-program P and every input X:

$$\langle Int\ P,\ X\rangle_B = \langle P\ X\rangle_A$$

Definition 2 (self-interpreter)
A self-interpreter for A is an A/A-interpreter.

Definition 3 (translator)
An M-program *Trans* is an A→B/M-translator if for every A-program P and every input X:

$$\langle\langle Trans\ P\rangle_M\ X\rangle_B = \langle P\ X\rangle_A$$

Definition 4 (generating extension)
An M-program Gen_P is a B/M-generating extension of an A-program P if for every input X, Y:

$$\langle\langle Gen_P\ X\rangle_M\ Y\rangle_B = \langle P\ X,\ Y\rangle_A$$

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. A self-interpreter interprets the same language it is written in. A generating extension allows to perfrom a computation in two stages, while the original program performs the same computation in one stage. The two-level generating extensions defined here can be generalized to multi-level generating extensions.[17)]

§3 Metacomputation

We refer to any process of simulating, analyzing and transforming programs by means of programs as *metacomputation*. Programs that carry out these tasks, are *metaprograms*. The metacomputation formulas used here are collectively referred to as *MST-formulas* (MST = metasystem transition[18)]).

3.1 Abstraction

To represent application expressions without specifying all data elements, we introduce *variables*. An application expression including variables, called *configuration*,[7)] is an abstraction which represents a set of application expressions. A variable ranges over the whole data domain. We use lower case names to denote variables. For instance, the following expression is a configuration where y is a variable:

$$\langle P\ X,\ y\rangle$$

3.2 Encoding

To represent configurations as data, we assume an injective mapping, called *metacoding*, from configurations into the data domain (necessary since variables are not elements of the data domain).* We are not interested in a specific way of metacoding and write a horizontal line above a configuration to denote its metacode. This allows a metacode-invariant representation, e.g. the following is a metacoded configuration:

$$\overline{\langle P\ X,\ y\rangle}$$

3.3 Metacomputation

It follows from our notation that $\langle Meta\ \overline{\langle P\ X,\ y\rangle}\rangle \Rightarrow Out$ denotes *metacomputation* on a metacoded configuration $\overline{\langle P\ X,\ y\rangle}$ by metaprogram *Meta*. This characterization of metacomputation states nothing about its concrete nature, except that it involves a metaprogram that operates on a metacoded configuration. For better readability, we move metacoded expression one line down for each metacoding (this two-dimensional notation, called *MST-schemes*, was suggested by V. Turchin):

* Metacoding in metacomputation corresponds to the Gödel numeration in logics where statements about a theory are encoded in the theory itself.

$$\langle Meta \underset{\langle P\ X,\ y\rangle}{\underline{\qquad\qquad}} \rangle \Rightarrow Out$$

3.4 Hierarchy of Metaprograms

When we abstract from a metacoded subexpression, we interrupt the horizontal line denoting the metacode. We assume that the metacode is compositional, i.e. changing a subcomponent does not entail a change in the enclosing metacode. For example,

$$\langle Meta \underset{\langle P\ \downarrow,\ y\rangle}{\underline{\quad x \quad}} \rangle$$

is a configuration where variable x abstracts from constant X. The distance between the variable and position $\downarrow$ defines how many times the value of a variable has to be metacoded upon substitution (here x's value needs to be metacoded once in order to obtain $\overline{X}$). The metacode of the above configuration can be passed, again, to a metaprogram.

Repeated use of abstraction, encoding, and metacomputation leads to a hierarchy of metaprograms. For example,

$$\langle Meta' \underset{\langle Meta \underset{\langle P\ \downarrow,\ y\rangle}{\underline{\;\; x \;\;}} \rangle}{\underline{\qquad\qquad\qquad}} \rangle \Rightarrow Out'$$

is a hierarchy of two metaprograms. The construction of each next level in a metasystem hierarchy is referred to as *metasystem transition*.[18] We speak of *self-application* if a metaprogram is applied to a copy of itself, i.e. if programs *Meta* and *Meta′* are identical.

§4 Equivalence Transformers

We are now in the position to define an important class of metaprograms, namely *equivalence transformers*, and to specify two transformation tasks. An equivalence transformer modifies the structure of programs with the purpose of optimizing some aspects of the programs performance while preserving the programs functionality. Two important types of equivalence transformers are *program specializers* and *program composers*. A specializer constructs a specialized program given a program P together with part of its input X. A composer transforms the composition of two programs P, Q into a single program. The meta-notation $C\sigma$ to denote the result of applying substitution σ to configuration C.

Definition 5 (equivalence transformer)
A program *Meta* is an equivalence transformer if for all substitutions $\sigma = \{x_1 \mapsto X_1, ..., x_n \mapsto X_n\}$ where $x_1, ..., x_n$ are all variables that occur free in a configuration C:*

* Assume *Meta* uses a fixed order for free variables, e.g. by occurrence from left to right.

$$\langle\langle Meta\ \overline{C}\rangle\ X_1 ... X_n\rangle = C\ \sigma$$

Definition 6 (program specializer)
An equivalence transformer *Spec* written in M is an A→B/M-specializer if for every A-program *P*, every input *X*, *Y*:

$$\langle\langle Spec \underset{\langle P\ X,\ y\rangle_{\mathrm{A}}}{\underline{\qquad\qquad}}\rangle_{\mathrm{M}}\ Y\rangle_{\mathrm{B}} = \langle P\ X,\ Y\rangle_{\mathrm{A}}$$

Definition 7 (program composer)
An equivalence transformer *Comp* written in M is an A→B/M-composer if for all A-programs *P*, *Q*, every input *X*, *Y*:

$$\langle\langle Comp \underset{\langle P\ \langle Q\ x\rangle_{\mathrm{A}},\ y\rangle_{\mathrm{A}}}{\underline{\qquad\qquad\qquad}}\rangle_{\mathrm{M}}\ X,\ Y\rangle_{\mathrm{B}} = \langle P\ \langle Q\ X\rangle_{\mathrm{A}},\ Y\rangle_{\mathrm{A}}$$

Although the above definitions state nothing about the quality of the transformers, we expect them to be non-trivial since the practical value of specialization and composition depends on their actual transformation power. A *non-trivial specializer* recognizes which of *P*'s computations can be precomputed at specialization time and which must be delayed until run-time, so as to yield an efficient specialized program. A *non-trivial composer* generates an efficient composition of *P*, *Q* by removing redundant computations, intermediate data structures, and other interface code which exact a cost at run-time. We shall not fix a specific transformation method and refer to Refs. 1), 2), and 4) for specialization methods and to Refs. 7), 8), 9), 10), and 19) for composition methods.

§5 Program Generation by Specialization

The three *Futamura projections* (FMP) tell us if we write a self-applicable specializer we get much more than just a specializer: we get the possibility to convert programs into generating extensions, and to produce a generator of generating extensions. We now fromalize the first two FMP as MST-schemes in order to compare them directly with the degeneration projections defined later. To focus on the essence of program generation, we assume that all specializers are source-to-source transformers written in the source language; for multi-language specialization see Ref. 20).

Let *Spec* be a source-to-source specializer and let *P* be a program with input *X*, *Y*. Assume that input *X* is available before input *Y*. Program *P* can be converted into an (efficient) generating extension using the 2nd FMP which follows from the 1st FMP.

1st FMP
Given *P*, *X* define a specialized program P_X such that

$$\langle P_X\ Y\rangle = \langle P\ X,\ Y\rangle \tag{1}$$

for all Y. For every P, X there exists a (trivial) program P_X as shown constructively by the definition

$$\mathbf{def}\ \langle P_X\ y\rangle \triangleq \langle P\ X,\ y\rangle$$

A (more efficient) program P'_X can be obtained by specializing the definition of P_X using *Spec*. The *1st FMP* is defined by

$$\langle Spec \underset{\langle P\ X,\ y\rangle}{\underline{\qquad\qquad}} \rangle \Rightarrow P'_X \tag{2}$$

According to Definition 6 we have for P'_X the equation:

$$\langle P'_X\ Y\rangle = \langle P\ X,\ Y\rangle \tag{3}$$

Program P'_X can often be optimized by *Spec* based on the input available at specialization time. The transformed program is faithful to the original program, but is often significantly faster. Optimization is achieved by changing the times at which computations are performed.

2nd FMP
Given P define a generating extension Gen_P such that

$$\langle\langle Gen_P\ X\rangle\ Y\rangle = \langle P'_X\ Y\rangle \tag{4}$$

for all X, Y. For every P there exists a (trivial) generating extension Gen_P by abstracting from constant X in the 1st FMP:

$$\mathbf{def}\ \langle Gen_P\ x\rangle \triangleq \langle Spec \underset{\langle P\ \bullet,\ y\rangle}{\underline{\quad x \quad}} \rangle$$

A (more efficient) generating extension Gen'_P can be obtained by specializing the definition of Gen_P using *Spec*. The *2nd FMP* is defined by

$$\langle Spec \underset{\langle Spec \underset{\langle P\ \bullet,\ y\rangle}{\underline{\quad x \quad}} \rangle}{\underline{\qquad\qquad\qquad}} \rangle \Rightarrow Gen'_P \tag{5}$$

The correctness of the projection follows from the correctness of the specializer. Note that program Gen'_P is obtained by self-appliation of *Spec*. According to Definition 6 we have for Gen'_P the equation:

$$\langle\langle Gen'_P\ X\rangle\ Y\rangle = \langle P'_X\ Y\rangle \tag{6}$$

The generating extension Gen'_P obtained by the 2nd FMP produces a specialized program often significantly faster than Gen_P that uses the 1st FMP because Gen'_P is a specializer specialized with respect to P. Using Gen'_P often speeds up specialization by a factor of three to four compared with Gen_P using the 1st FMP.[21)]

A well-known application of the 2nd FMP is the conversion of an interpreter into a translator.[6] Interpreted programs run typically an order of magnitude slower than those which are translated: a difference large enough to be worth reducing by converting an interpreter into a translator.

Example 1 (translator generation)
Let *IntBC* be a B/C-interpreter. Given a C→C/C-specializer *Spec*, the B/C-interpreter can be converted into a B→C/C-translator *TransBC* such that $\langle\langle TransBC\ P\rangle_C\ D\rangle_C = \langle IntBC\ P,\ D\rangle_C$.

$$\langle Spec \underset{\langle Spec \underset{\langle IntBC\ \overset{\downarrow}{\bullet},\ d\rangle_C}{\underline{\qquad p \qquad}} \rangle_C}{\underline{\qquad\qquad\qquad\qquad}} \rangle_C \Rightarrow TransBC \qquad \square$$

§6 Program Degeneration by Composition

Consider the reverse goal: turning a generating extension into an ordinary program. We call this the *degeneration* of a generating extension. In other words, a two-stage computation is turned into a one-stage computation.

Let *Comp* be a source-to-source composer and let Gen_P be a source-to-source generating extension. Suppose a self-interpreter *Sint* for the source language is given. We now present the 1st *degeneration projection* (DGP) for finding an (efficient) one-stage program given the generating extension.

1st DGP
Given Gen_P define a program P such that

$$\langle P\ X,\ Y\rangle = \langle\langle Gen_P\ X\rangle\ Y\rangle \qquad (7)$$

for all X, Y. For every generating extension Gen_P there exists a (trivial) program P as shown constructively by the definition

$$\textbf{def}\ \langle P\ x,\ y\rangle \triangleq \langle Sint\ \langle Gen_P\ x\rangle,\ y\rangle$$

where P's computation is performed in two stages: first by producing a specialized program using Gen_P, then interpreting the new program with *Sint*. For practical reasons, such a trivial degeneration of Gen_P is uninteresting.

A (more efficient) program P' can be obtained by applying the composer *Comp* to P's definition in order to remove the computational overhead caused by the generation and subsequent interpretation of a specialized program. The *1st DGP* is defined by

$$\langle Comp \underset{\langle Sint\ \langle Gen_P\ x\rangle,\ y\rangle}{\underline{\qquad\qquad\qquad}} \rangle \Rightarrow P' \qquad (8)$$

The correctness of P' follows from the correctness of the composer. According to Definitions 2 and 7 we have for P' the equation:

$$\langle P'\ X,\ Y\rangle = \langle\langle Gen_P\ X\rangle\ Y\rangle \qquad (9)$$

Similar to specialization, we expect that a non-trivial composer generates a program P' that is significantly faster than the (trivial) P defined above. On the other hand, even though *Comp* may be powerful enough to reduce all interpretive overhead caused by *Sint*, P' may not run faster than the program from which Gen_P was generated, e.g. by the 2nd FMP, unless this program is inefficient and can be optimized.

So if we are given a composer and a self-applicable specializer we have the possibility to convert generating extensions into ordinary programs and vice versa. The degeneration scheme can be applied to all generating extensions, including those shown in Fig.1.

$$\begin{tikzcd} P \arrow[r, shift left, "\text{generation by specialization}"'] & Gen_P \arrow[l, shift left, "\text{degeneration by composition}"'] \end{tikzcd}$$

An application of the 1st DGP is the conversion of a translator into an interpreter.[14] Although interpretive language definitions are often preferable, e.g. when a new language is defined from scratch, it may be convenient to define language extensions by translation into elementary constructs (cf. bootstrapping compilers, macro expanders). In this situation the need for converting a translator into an interpreter may arise.

Example 2 (translator degeneration)
Let *TransBC* be a $B\to C/C$-translator, and *Sint* be a self-interpreter for C. Given a $C\to C/C$-composer *Comp*, the $B\to C/C$-translator can be converted into a B/C-interpreter *IntBC'* such that $\langle\langle TransBC\ P\rangle_C\ X\rangle_C = \langle IntBC'\ P,\ X\rangle_C$.

$$\langle Comp_{\overline{\langle Sint\ \langle TransBC\ p\rangle_C,\ x\rangle_C}}\rangle_C \Rightarrow IntBC' \qquad \square$$

Note that the degeneration of a generating extension does *not*, in contrast to the generation of a generating extension, require self-application, only a single metasystem level. Degeneration is conceptually simpler, but technically not necessarily easier than the generation of a generating extension by self-application of a specializer. In fact, all three FMP give good results in practice,[4] while the computational realization of the DGP is still an open problem.

§7 A Generator of Generating Extensions

We review the 3rd FMP which allows us to produce a stand-alone tool for program generation, namely a *generator of generating extensions*. It can be obtained by double self-application of a specializer.

3rd FMP
Let Gen'_P be the generating extensions of P defined in Equation (6). Define a generator of generating extension *Gegen* such that

$$\langle\langle Gegen\ P\rangle\ X\rangle = \langle Gen'_P\ X\rangle \qquad (10)$$

for all *P*, *X*. For every specializer *Spec* there exists a (trivial) generator *Gegen* by abstracting from *P* in the 2nd FMP:

$$\mathbf{def}\ \langle \mathit{Gegen}\ p\rangle \triangleq \begin{array}{l}\langle \mathit{Spec}\_\_\_\_\_\ p\ \_\_\_\_\_\rangle\\ \qquad\langle \mathit{Spec}\_\,|\,\_x\_\_\_\rangle\\ \qquad\qquad\langle \bullet\ \bullet,\ y\rangle\end{array}$$

A (more efficient) program *Gegen′* can be obtained by specializing the definition of *Gegen* using *Spec* for the third time. The *3rd FMP* is defined by

$$\begin{array}{l}\langle \mathit{Spec}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\rangle \Rightarrow \mathit{Gegen}'\\ \qquad\langle \mathit{Spec}\_\_\_\_\_\ p\ \_\_\_\_\_\rangle\\ \qquad\qquad\langle \mathit{Spec}\_\,|\,\_x\_\_\_\rangle\\ \qquad\qquad\qquad\langle \bullet\ \bullet,\ y\rangle\end{array} \tag{11}$$

According to Definition 6 we have for *Gegen′* the equation:

$$\langle\langle \mathit{Gegen}'\ P\rangle\ X\rangle = \langle \mathit{Gen}'_P\ X\rangle \tag{12}$$

Example 3 (translator generation revisited)
Returning to Example 1, the B/C-interpreter *IntBC* can be transformed into a translator *TransBC* by *Gegen*:

$$\langle \mathit{Gegen}\ \mathit{IntBC}\rangle \Rightarrow \mathit{TransBC} \qquad \square$$

§8 A Degenerator of Generating Extensions

We now present a stand-alone tool for program degeneration, namely a *degenerator of generating extensions*, and show how it can be obtained from a specializer and a composer. This section introduces the 2nd and 3rd DGP.

2nd DGP
Let *P′* be the program defined in Equation (9). Define a degenerator of generating extensions *Degen* such that

$$\langle\langle \mathit{Degen}\ \mathit{Gen}_P\rangle\ X,\ Y\rangle = \langle P'\ X,\ Y\rangle \tag{13}$$

for all *Gen*$_P$ *X*, *Y*. Given self-interpreter *Sint*, there exists a (trivial) degenerator *Degen* for every composer *Comp* by abstracting from *Gen*$_P$ in the 1st DGP:

$$\mathbf{def}\ \langle \mathit{Degen}\ \mathit{gen}\rangle \triangleq \begin{array}{l}\langle \mathit{Comp}\_\_\_\_\_\ \mathit{gen}\ \_\_\_\_\_\rangle\\ \qquad\langle \mathit{Sint}\ \langle\bullet\ x\rangle,\ y\rangle\end{array}$$

A (more efficient) degenerator *Degen′* can be obtained by applying specializer *Spec* to *Degen*'s definition in order to specialize the composer with respect to *Sint*. The *2nd DGP* is defined by

$$\begin{array}{l}\langle \mathit{Spec}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\rangle \Rightarrow \mathit{Degen}'\\ \qquad\langle \mathit{Comp}\_\_\_\_\_\ \mathit{gen}\ \_\_\_\_\_\rangle\\ \qquad\qquad\langle \mathit{Sint}\ \langle\bullet\ x\rangle,\ y\rangle\end{array} \tag{14}$$

According to Definition 6 we have for *Degen′* the equation:

$$\langle\langle \mathit{Degen}'\ \mathit{Gen}_P\rangle\ X,\ Y\rangle = \langle P'\ X,\ Y\rangle \tag{15}$$

Note that the 2nd DGP does not use a self-applicable specializer as does the 2nd FMP, but a metasystem structure consisting of a composer and a specializer. The 2nd DGP exhibits the underlying metasystem structure of the degenerator problem and tells us what can be achieved given powerful enough composers and specializes. It allows us to identify certain abstract properties needed to obtain a degenerator. In practice, a transformer may, of course, be powerful enough to serve both as a non-trivial specializer and a non-trivial composer.[7]

Example 4 (degenerator)
Continuing Example 2, we can convert the translator *TransBC* into an interpreter *IntBC′* using the degenerator *Degen*:

$$\langle \mathit{Degen}\ \ \mathit{TransBC}\rangle \Rightarrow \mathit{IntBC}' \qquad \square$$

3rd DGP
Finally, one can obtain a generator of degenerators as follows. Define a generator of degenerators *Gedegen* such that

$$\langle\langle \mathit{Gedegen}\ \ \mathit{Sint}\rangle\ \ \mathit{Gen}_P\rangle = \langle \mathit{Degen}'\ \ \mathit{Gen}_P\rangle \qquad (16)$$

for all *Sint*, *Gen*$_P$. Given *Sint*, *Gedegen* produces a new degenerator which is then applied to the generating extension *Gen*$_P$. For every *Comp*, *Spec* there exists a (trivial) degenerator generator *Gedegen* by abstracting from *Sint* in the 2nd DGP:

$$\begin{array}{l} \textbf{def}\ \langle \mathit{Gedegen}\ \ \mathit{sint}\rangle \triangleq \langle \mathit{Spec}\_\_\_\_\_\mathit{sint}\_\_\_\_\_\_\rangle \\ \qquad\qquad\qquad\qquad\qquad \langle \mathit{Comp}\_|\_\mathit{gen}\_\_\_\_\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad \langle\bullet\ \langle\bullet\ x\rangle,\ y\rangle \end{array}$$

A (more efficient) degenerator generator *Gedegen′* can be obtained by specializing *Gedegen*'s definition using *Spec*. The *3rd DGP* is defined by

$$\begin{array}{l} \langle \mathit{Spec}\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\rangle \Rightarrow \mathit{Gedegen}' \\ \qquad \langle \mathit{Spec}\_\_\_\_\_\mathit{sint}\_\_\_\_\_\_\rangle \\ \qquad\qquad \langle \mathit{Comp}\_|\_\mathit{gen}\_\_\_\_\rangle \\ \qquad\qquad\qquad \langle\bullet\ \langle\bullet\ x\rangle,\ y\rangle \end{array} \qquad (17)$$

According to Definition 7 we have for *Gedegen′* the equation:

$$\langle\langle \mathit{Gedegen}'\ \ \mathit{Sint}\rangle\ \ \mathit{Gen}_P\rangle = \langle \mathit{Degen}'\ \ \mathit{Gen}_P\rangle \qquad (18)$$

A degenerator generator may be used to produce different degenerators by modifying the self-interpreter. This is similar to the methods for achieving different transformation effects by instrumenting interpreters, called the *interpretive approach*.[22,20] In general, *Sint* need not even be a self-interpreter but can be any interpreter as long as it is written in the source language of the composer. This may be useful for producing degenerators with different source languages from a generic specializer and a generic composers, a method which is similar to the generation of specializers with different source languages from a generic specializer.[20]

§9 A Degenerating Interpreter

We have seen that a degenerator can be obtained by specializing a composer. It is remarkable that there exists yet another method to obtain a degenerator, namely by viewing a degenerator *Degen* as a generating extension of a degenerating interpreter *Dint*. This suggests the existence of a generation/ degeneration relation between these two programs which allows us to obtain a degenerator from a degenerating interpreter using the FMP, and to degenerate a degenerator using the DGP.

$$\boxed{\begin{tikzcd} Dint \arrow[r, shift right, "\text{generation}"'] & Degen \arrow[l, shift right, "\text{degeneration}"'] \end{tikzcd}}$$

9.1 Degenerating Interpreter

Consider an interpreter of generating extensions, call it *Dint*, which satisfies the equation

$$\langle Dint\ Gen_P,\ X,\ Y\rangle = \langle P\ X,\ Y\rangle \tag{19}$$

for all generating extensions Gen_P and all input X, Y. The interpreter *Dint* returns the same result when applied to P's generating extension Gen_P and input X, Y as program P applied to X, Y. The existence of *Dint*, which we call *degenerating interpreter*, is shown constructively by the definition

$$\textbf{def}\ \langle Dint\ gen,\ x,\ y\rangle \triangleq \langle Sint\ \langle Sint\ gen\ x\rangle,\ y\rangle$$

A consequence of Equations (15) and (19) is the equation

$$\langle Dint\ Gen_P,\ X,\ Y\rangle = \langle\langle Degen\ Gen_P\rangle\ X,\ Y\rangle \tag{20}$$

which suggests that a *degenerator is a generating extension of a degenerating interpreter*, just as translators and specializers are generating extensions of ordinary interpreters (considering Gen_P as 'first' and X, Y as 'second' argument of *Dint*).

In fact, *Dint* can be seen as a non-standard semantics of generating extensions.[23)] It provides the semantics of Gen_P's implementation language, as well as the semantics of Gen_P's target language.

9.2 Generation of a Degenerator

Viewing a degenerator as a generating extension of a degenerating interpreter *Dint* means that a degenerator *Degen'* can be obtained from *Dint* using the program generator *Gegen* from Section 7:

$$\langle Gegen\ Dint\rangle \Rightarrow Degen'$$

This is remarkable because one does not need a "universal" composer, but can resolve the composition problem already in the (manual) construction of *Dint*. In other words, conventional specialization is sufficient to obtain a degenerator. It may be easier to write *Dint* than to design a general composer that specializes well with respect to (arbitrary) interpreters. This is one more example of generating a transformer from an interpreter (cf. Section 8).

9.3 Degeneration of a Degenerator

On the other hand, a degenerating interpreter can be obtained by degenerating a degenerator:

$$\langle \mathit{Degen}\ \mathit{Degen}' \rangle \Rightarrow \mathit{Dint}'$$

It can easily be verified that *Dint'* is a degenerating interpreter by combining Equations (19) and (15):

$$\langle \mathit{Dint}'\ \mathit{Gen}_P,\ X,\ Y \rangle = \langle \mathit{Dint}\ \mathit{Gen}_P,\ X,\ Y \rangle$$

A possible application is testing of a new degenerator *Degen'* by verifying the correctness of *Dint'*.

9.4 Self-Degeneration

Finally, as a special case, *Degen* can be applied to a copy of itself. We call this *self-degeneration*:

$$\langle \mathit{Degen}\ \mathit{Degen} \rangle \Rightarrow \mathit{Dint}''$$

This suggests an interesting test for degenerators, namely that a degenerator must be strong enough to degenerate itself into an efficient degenerating interpreter (this corresponds to the requirement[4] that a specializer is self-applicable if it is strong enough to produce an efficient specialization of itself).

§10 An Application of Incremental Degeneration

It is known that a specializer can be seen as a generating extension of an interpreter, and that a generator of generating extensions can be seen as a generating extension of a specializer.[20] We conclude the discussion with an example of *incremental degeneration*, namely the degeneration of a generator *Gegen* into a specializer *Mix*, and the degeneration of *Mix* into an interpreter *Int2*. The generation direction is described by the specializer projections,[20] and practical results have been achieved for the conversion from *Int2* to *Mix*, and from *Mix* to *Gegen*.[17]

$$\begin{tikzcd} \mathit{Int2} \arrow[r, shift right, "\text{generation}"'] & \mathit{Mix} \arrow[l, shift right, "\text{degeneration}"'] \arrow[r, shift right, "\text{generation}"'] & \mathit{Gegen} \arrow[l, shift right, "\text{degeneration}"'] \end{tikzcd}$$

10.1 Generator Degeneration

Let *Gegen* be a generator of generating extensions as specified by Equation (12). Consider equation

$$\langle\langle Gegen\ P\rangle\ X\rangle = \langle Mix\ P, X\rangle \tag{21}$$

which follows from the 3rd FMP and the definition of a specializer *Mix*:

$$\textbf{def}\ \langle Mix\ p, x\rangle \triangleq \langle Spec\_\underset{\langle\bullet}{\overset{}{p}}\ \ \underset{\bullet,\ y\rangle}{x}\underline{\quad\ }\rangle$$

The specializer *Mix* returns the same result when applied to program *P* and its first input *X* as *P*'s generating extensions (produced by generator *Gegen* from *P*) applied to *X*. The equation suggests, cf. Equation (9), that a generator *Gegen* is a generating extension of a specializer *Mix* and, hence, both can be transformed into each other by generation and degeneration:

$$\langle Degen\ \ Gegen\rangle \Rightarrow Mix'$$
$$\langle Gegen\ \ Mix'\rangle \Rightarrow Gegen'$$

An interesting property is that *Gegen* is *self-generating*, i.e. $\langle Gegen\ \ Mix\rangle \Rightarrow Gegen$, if *Gegen* is the result of the 3rd FMP using the same specializer *Spec* as in the definition of *Mix* above (this property was first proven in Ref. 2)).

10.2 Specializer Degeneration

Let *Mix* be the specializer defined above. Consider equation

$$\langle\langle Mix\ P, X\rangle\ Y\rangle = \langle Int2\ P, X, Y\rangle \tag{22}$$

which follows from the definition of *Mix* and the specification of an interpreter *Int2* of two-argument programs:

$$\langle Int2\ P, X, Y\rangle = \langle P\ X, Y\rangle$$

The interpreter *Int2* applied to program *P* and input *X*, *Y* returns the same result as *P*'s residual program (produced by *Mix* from *P*, *X*) applied to the remaining input *Y*. The equation suggests, cf. Equation (9), that a specializer *Mix* is a generating extension of an interpreter *Int2* and that both can be generated/degenerated into each other:

$$\langle Degen2\ \ Mix\rangle \Rightarrow Int2'$$
$$\langle Gegen2\ \ Int2'\rangle \Rightarrow Mix'$$

where *Degen2* and *Gegen2* handle the arguments *P*, *X* together as 'first' argument.

§11 Two Tests for Generators and Degenerators

As noted earlier (Section 4), the quality of the programs resulting from generation and degeneration depends on the transformations and strategies employed in the corresponding transformer.

The development of transformation tools requires certain test procedures to ensure that the tools are correct and 'strong enough'. If a collection of tools allows us to perform some operations and their inverse, we have a possibility to check their correctness w.r.t. each other. In our case: a program supplied to a specializer gives a generating extension of the program, which degenerated by a composer and a self-interpreter provides a program equivalent to the original one. The power of program transformers can also be tested this way: if the resulting program is at least as good as the original one, then the tools are strong enough w.r.t. this test.

The projections for generation and degeneration suggest excellent tests for program specializers and program composers. From the equations of generator *Gegen* (12) and degenerator *Degen* (15) it follows that they are inverse to each other:

$$\langle Degen\ \langle Gegen\ P\rangle\rangle \Rightarrow P'$$
$$\langle Gegen\ \langle Degen\ Gen_P\rangle\rangle \Rightarrow Gen'_P$$

This suggests two natural goals: (i) that P' be at least as efficient as P and (ii) that Gen'_P be at least as efficient as Gen_P. Achieving the former goal implies that a degenerator *Degen* can remove all overhead introduced by a generator; achieving the second goal implies that the generator *Gegen* reintroduces effective staging in any degenerated program. In each case, one could reasonably say that *Degen* or *Gegen* is "optimal".

The following tests for *Degen* and *Gegen* are stated w.r.t. a given generator and degenerator, respectively. It would be too much to ask that the *Degen* or *Gegen* in question yields optimal degeneration and generation for *all* possible generators and degenerators. To describe the tests more precisely, let $t_P(D)$ be the time to compute $\langle P\ D\rangle$.

Definition 8 (degeneration test)
A degenerator *Degen* is optimal w.r.t. generator *Gegen* provided

$$t_{P'}(X,\ Y) \le t_P(X,\ Y)$$

for all P, X, Y, where

$$\langle Degen\ \langle Gegen\ P\rangle\rangle \Rightarrow P'$$

Definition 9 (generation test)
A generator *Gegen* is optimal w.r.t. degenerator *Degen* provided

$$t_{Gen'_P}(X) \le t_{Gen_P}(X)$$

for all Gen_P, X, where

$$\langle Gegen\ \langle Degen\ Gen_P\rangle\rangle \Rightarrow Gen'_P$$

§12 Summary

Figure 2 summarizes the projections for program generation and degeneration discussed in Sections 5 to 8. Let all programs be written in language C (hence, we omit the languages indices). All projections can be performed given two C→C/C-transformers: a specializer *Spec* and a composer *Comp*.

The following equations hold for every program P, every generating extension Gen_P of P, every C-self-interpreter *Sint*, and every input X, Y. Composing Equations (3), (6), and (12), we obtain:

$$\begin{aligned}\langle P\ X,\ Y\rangle &= \langle P'_X\ Y\rangle\\ &= \langle\langle Gen'_P\ X\rangle\ Y\rangle\\ &= \langle\langle\langle Gegen'\ P\rangle\ X\rangle\ Y\rangle \qquad (23)\end{aligned}$$

Composing Equations (9), (15), and (18), we obtain:

$$\begin{aligned}\langle P\ X,\ Y\rangle &= \langle P'\ X,\ Y\rangle\\ &= \langle\langle Degen'\ Gen_P\rangle\ X,\ Y\rangle\\ &= \langle\langle\langle Gedegen'\ Sint\rangle\ Gen_P\rangle\ X,\ Y\rangle \qquad (24)\end{aligned}$$

Generation		*Result*
1st FMP	$\langle Spec\,\underline{\qquad\qquad}\rangle \Rightarrow P'_X$ $\langle P\ X,\ y\rangle$	specialized program
2nd FMP	$\langle Spec\,\underline{\qquad\qquad\qquad}\rangle \Rightarrow Gen'_P$ $\langle Spec\,\underline{\quad x\quad}\rangle$ $\langle P\ ●,\ y\rangle$	generating extension
3rd FMP	$\langle Spec\,\underline{\qquad\qquad\qquad\qquad}\rangle \Rightarrow Gegen'$ $\langle Spec\,\underline{\qquad p\qquad}\rangle$ $\langle Spec\,\underline{\ \ x\quad}\rangle$ $\langle ●\ ●,\ y\rangle$	generator of gen. extensions

Degeneration		*Result*
1st DGP	$\langle Comp\,\underline{\qquad\qquad\qquad}\rangle \Rightarrow P'$ $\langle Sint\ \langle Gen_P\ x\rangle,\ y\rangle$	program
2nd DGP	$\langle Spec\,\underline{\qquad\qquad\qquad\qquad}\rangle \Rightarrow Degen'$ $\langle Comp\,\underline{\qquad gen\qquad}\rangle$ $\langle Sint\ \langle ●\ x\rangle,\ y\rangle$	degenerator
3rd DGP	$\langle Spec\,\underline{\qquad\qquad\qquad\qquad\qquad}\rangle \Rightarrow Gedegen'$ $\langle Spec\,\underline{\qquad sint\qquad}\rangle$ $\langle Comp\,\underline{\ \ gen\qquad}\rangle$ $\langle ●\ \langle ●\ x\rangle,\ y\rangle$	generator of degenerators

Fig. 2 MST-schemes for generation and degeneration.

Figure 3 summarizes applications of generation and degeneration. Let *Gegen* and *Degen* be a generator and degenerator of generating extensions, respectively. The programs produced as output, e.g. P' in case (a), are marked with a prime because, in general, they are not textually identical to the programs given as input, e.g. P (but they are computationally equivalent). Beside the application to the examples in Fig. 1, summarized in the generic case (a), the following four application are note-worthy:

	Generation	Degeneration
a	$\langle Gegen\ P \rangle \Rightarrow Gen_P$	$\langle Degen\ Gen_P \rangle \Rightarrow P'$
b	$\langle Gegen\ Int \rangle \Rightarrow Trans$	$\langle Degen\ Trans \rangle \Rightarrow Int'$
c	$\langle Gegen2\ Int2 \rangle \Rightarrow Mix$	$\langle Degen2\ Mix \rangle \Rightarrow Int2'$
d	$\langle Gegen\ Mix \rangle \Rightarrow Gegen$	$\langle Degen\ Gegen \rangle \Rightarrow Mix'$
e	$\langle Gegen\ Dint \rangle \Rightarrow Degen$	$\langle Degen\ Degen \rangle \Rightarrow Dint'$

Fig. 3 Summary of generator and degenerator applications.

(b) the conversion between an interpreter *Int* and a translator *Trans* (Sections 5 and 6),

(c) the conversion between an interpreter *Int2* for two-argument programs and a specializer *Mix* (Section 10),

(d) the conversion between a specializer *Mix* and a generator of generating extensions *Gegen* (Section 10), and

(e) the conversion between a degenerating interpreter *Dint* and a degenerator *Degen* (Section 9).

The projections presented in this paper can be generalized in two directions without changing the underlying principles:

- *Multi-language transformers*. The projections for generation and degeneration were formulated assuming source-to-source transformers. It is clear that they can be generalized to multi-language transformers for appropriate languages; for multi-language specialization see Ref. 20).
- *Multi-level generating extensions*. The 1st DGP and the 2nd FMP are the basic building blocks for a class of formulas that generate and degenerate multi-level generating extensions (Gen'_P is a two-level generating extension). All multi-level formulas can be performed by incremental generation and degeneration (Section 10), although in practice a direct conversion may have certain technical advantages. A multi-level generator based on off-line partial evaluation was designed and implemented.[17)]

§13 Conclusion

While certain aspects of the software development process are likely never to be fully automated, the technology of building tools that treat programs as data objects is very powerful. This paper studied two fundamental operations on programs, namely program specialization and program composition, which are both equivalence transformations, and showed that both are essential for a wide spectrum of transformation tasks. Function specialization and function composition are fundamental operations in Mathematics, and, not surprisingly, the same is true for Computer Science. We showed that their full realization in practice may have far reaching consequences for the automatic development of software by software.

As often, particular methods can solve specific transformation tasks better and more efficiently. Nevertheless, the main challenge remains: the development of transformation methods that optimize not just special, but general cases of equivalence transformation problems. Program specialization, one of the best developed methods, was the first step. We believe that program composition is the next step on the research agenda. The transformation problems presented in this paper may serve as test cases for existing methods, and as challenging problems for future research.

A widely known application of the Futamura projections is the conversion of interpreters into translators. Respectively, the degeneration projections convert translators into interpreters. The need of the latter application may not be so evident as that of the former, since interpreters are usually easier to construct than translators and translated programs are faster than interpreted programs. However, there exist cases where the conversion of translators to interpreters may be necessary.

(1) Often it is easier to read and verify an interpretive program than a program transformer. Degeneration can convert a transformer into an interpreter. Verifying the interpreter can guarantee the correctness of certain aspects of the transformer (provided the degenerator is correct, which has to be shown only once). For instance, this can help to verify the correctness of a hand-written translator.
(2) Language definition by interpretation is not always simpler than by translation. If a language is extended gradually, defining language extensions by translation into more elementary constructs is often easier than writing a new interpretive definition for the full language (e.g., macrosystems exploit this feature).
(3) Another remarkable example, although not from a programming perspective, are formal languages developed by mathematics: mathematical definitions are, in general, nothing else but translative definitions, similar to macros. (A corresponding approach to manipulation of mathematical theories is discussed in Refs. 7) and 14)).

Acknowledgements

Thanks to the anonymous referees for their constructive comments.

References

1) Ershov, A. P., "On the Essence of Compilation," in *Formal Description of Programming Concepts* (E. J. Neuhold, ed.), North-Holland, pp. 391-420, 1978.
2) Futamura, Y., "Partial Computation of Programs," in *RIMS Symposia on Software Science and Engineering* (E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, eds.), Springer-Verlag, pp. 1-35, 1983.
3) Bjørner, D., Ershov, A. P., and Jones, N. D. (eds.), *Paritial Evaluation and Mixed*

Computation, North-Holland, 1988.
4) Jones, N. D., Gomard, C. K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.
5) Danvy, O., Glück, R., and Thiemann, P. (eds.), *Partial Evaluation, volume 1110 of Lecture Notes in Computer Science*, Springer-Verlag, 1996.
6) Futamura, Y., "Partial Evaluation of Computing Process—An Approach to a Compiler-Compiler," *Systems, Computers, Controls, 2, 5*, pp. 45-50, 1971.
7) Turchin, V. F., "The Concept of a Supercompiler," *Transactions on Programming Languages and Systems, 8, 3*, pp. 292-325, 1986.
8) Wadler, P., "Deforestation: Transforming Programs to Eliminate Trees," *Theoretical Computer Science, 73*, pp. 231-248, 1990.
9) Fegaras, L., Sheard, T., and Zhou, T., "Improving Programs Which Recurse over Multiple Inductive Structures," in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, pp. 21-32, 1994.
10) Takano, A. and Meijer, E., "Shortcut Deforestation in Calculational Form," in *Conference on Functional Programming Languages and Computer Architecture*, ACM Press, pp. 306-313, 1995.
11) Turchin, V. F., "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)*, pp. 31-42, 1972. [In Russian]
12) Bratman, H., "An Alternate Form of the 'UNCOL Diagram,'" *Communications of the ACM, 4, 3*, p. 142, 1961.
13) Earley, J. and Sturgis, H., "A Formalism for Translator Interactions," *Communications of the ACM, 13, 10*, pp. 607-617, 1970.
14) Glück, R. and Klimov, A. V., "Metasystem Transition Schemes in Computer Science and Mathematics," *World Futures, 45*, pp. 213-243, 1995.
15) Glück, R., "On the Mechanics of Metasystem Hierarchies in Program Transformation," in *Logic Program Synthesis and Transformation, volume 1048 of Lecture Notes in Computer Science* (M. Proietti, ed.), Springer-Verlag, pp. 234-251, 1996.
16) Turchin, V. F., "Metacomputation : Metasystem Transitions plus Supercompilation," in *Partial Evaluation, volume 1110 of Lecture Notes in Computer Science* (O. Danvy, R. Glück, and P. Thiemann, ed.), Springer-Verlag, pp. 481-509, 1996.
17) Glück, R. and Jørgensen, J., "Efficient Multi-Level Generating Extensions for Program Specialization," in *Programming Languages: Implementations, Logics and Programs (PLILP'95)* (M. Hermenegildo and S. D. Swierstra, eds.), *volume 982 of Lecture Notes in Computer Science*, Springer-Verlag, pp. 259-278, 1995.
18) Turchin, V. F., *The Phenomenon of Science*, Columbia University Press, New York, 1977.
19) Tofte, M., *Compiler Generators, volume 19 of EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1990.
20) Glück, R., "On the Generation of Specializers," *Journal of Functional Programming, 4, 4*, pp. 499-514, 1994.
21) Bondorf, A., "Automatic Autoprojection of Higher Order Recursive Equations," *Science of Computer Programming, 17, 1-3*, pp. 3-34, 1991.
22) Turchin, V. F., "Program Transformation with Metasystem Transitions," *Journal of Functional Programming, 3, 3*, pp. 283-313, 1993.
23) Abramov, S. M., *Metavychislenija i ikh prilozhenija (Metacomputation and its Applications)*, Nauka, Moscow, 1995. [In Russian]

Robert Glück: He is an associate professor of Computer Science at the University of Copenhagen. He received his M. Sc. and Ph. D. degrees in 1986 and 1991 from the University of Technology in Vienna, where he also worked as assistant professor. He was research assistant at the City University of New York and twice received the Erwin-Schrödinger-Fellowship of the Austrian Science Foundation. His research interests include advanced programming languages, theory and practice of program transformation, and metaprogramming.

Andrei V. Klimov: He is a senior researcher at the Keldysh Institute of Applied Mathematics, the Russian Academy of Sciences. After receiving his M.Sc. degree from Moscow State University in 1976, he worked in a research institute in electronic industry, where developed software for microprocessors. Since 1983, when he changed to Keldysh Institute, his main research interests are functional programming, theory and methods of program transformation and optimization, and their application to practical programming languages and problems.