Is There a Fourth Futamura Projection?

Robert Glück *

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen, Denmark
glueck@acm.org

Abstract

The three classic Futamura projections stand as a cornerstone in the development of partial evaluation. The observation by Futamura [1983], that compiler generators produced by his third projection are self-generating, and the insight by Klimov and Romanenko [1987], that Futamura's abstraction scheme can be continued beyond the three projections, are systematically investigated, and several new applications for compiler generators are proposed. Possible applications include the generation of quasi-online compiler generators and of compiler generators for domain-specific languages, and the bootstrapping of compiler generators from program specializers. From a theoretical viewpoint, there is equality between the class of self-generating compiler generators and the class of compiler generators produced by the third Futamura projection. This exposition may lead to new practical applications of compiler generators, as well as deepen our theoretical understanding of program specialization.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Processors—compilers, interpreters; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—partial evaluation

General Terms Languages, Theory

Keywords Bootstrapping, Cogen approach, Compiler generators, Domain-specific languages, Futamura projections, Generator self-generation, Program specialization, Self-application.

1. Introduction

The three Futamura projections stand as a cornerstone in the development of partial evaluation [13, 14]. Practical specializers that can perform all three Futamura projections and that can automatically convert programs into non-trivial generating extensions and compiler generators have been built for realistic programming languages such as Scheme, Prolog, and C (*e.g.*, [1, 5, 38, 39]). Various practical applications of program specialization have been described and the usefulness of program specialization for software development, including domain-specific languages, has been demonstrated (*e.g.*, [31, 43]).

Based on this experience, the modern approach to the specialization of programs is to write compiler generators directly instead of writing self-applicable specializers (*e.g.*, [2, 3, 6, 9, 21, 27, 36, 41, 44, 45]). While this *cogen approach*, of hand-writing compiler generators based on partial evaluation principles, has shown its practical value, it prevents the use of the power and flexibility of different self-application schemes, such as combining online and offline specializers. Despite of more than two decades of practical experience with compiler generators in the area of partial evaluation, some fundamental properties and potential applications of these powerful and universal tools for program generation have not been fully utilized or well understood.

Among the basic questions are: what comes beyond the three Futamura projections, whether there exists a fourth projection, and whether there is any practical interest in this at all. This theoretical paper aims at addressing some of these open issues. Futamura's observation [15], that compiler generators produced by his third projection are *self-generating*, and Klimov and Romanenko's important insight [33], that *Futamura's abstraction scheme* can be continued beyond the three classical projections, are systematically investigated and extended. In addition, several new applications for compiler generators are proposed. The main goal is to show that these two observations may be of practical interest if a compiler generator is applied to different specializers.

Potential applications include (1) the generation of *compiler generators for domain-specific languages* from non-self-applicable specializers, (2) the generation of *quasi-online compiler generators* from offline compiler generators, and (3) alternatives to computing the Futamura projections by a *bootstrapping process*. On the theoretical side, (4) there is an equality between the *class of self-generating compiler generators* and the *class of compiler generators produced by the third Futamura projection*. This exposition may lead to new applications of compiler generators and may deepen the theoretical understanding of program specialization.

The paper is organized as follows. After introducing notation and definitions (Sect. 2), the three classic Futamura projections are reviewed (Sect. 3), and Futamura's self-generating compiler generators are examined (Sect. 4). Projections are developed by three more abstraction steps, showing the possibility of compiler-generator bootstrapping (Sect. 5). Then, the general scheme for generating compiler generators is presented (Sect. 6) and an application to domain-specific languages is proposed (Sect. 7). Finally, we discuss related work (Sect. 8) and present our conclusions (Sect. 9).

We assume that readers are familiar with the basics of partial evaluation, *e.g.*, as presented in the book by Jones *et al*. [28, Part II].

* Part of this work was performed while the author was visiting the National Institute of Informatics (NII), Tokyo. This work was also supported by the Danish Natural Science Research Council.

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

2. Definitions and Notation

This section introduces the notation and terminology used in this paper, which are fairly standard for readers familiar with partial evaluation. The notation is adapted from Jones *et al*. [28]. Readers familiar with partial evaluation may skip this section and proceed to Sect. 3.

2.1 Notation

A programming language L consists of a set of programs $P_{\mathtt{L}}$, a set of data D, and a semantics function $[\![\_]\!]_{\mathtt{L}} : P_{\mathtt{L}} \to (D \rightharpoonup D)$, which assigns to each program $\mathtt{p} \in P_{\mathtt{L}}$ a partial input-output function $[\![\mathtt{p}]\!]_{\mathtt{L}} : D \rightharpoonup D$. We assume that D includes all L-programs, $P_{\mathtt{L}} \subseteq D$, and all lists, $(\mathtt{d}_1, \ldots, \mathtt{d}_n) \in D$ where $\mathtt{d}_i \in D$. Taking all L-programs from D is convenient when dealing with meta-programs that input and output programs. Without loss of generality, we assume that all languages utilize the same D. Equality ($=$) denotes strong equality: either both sides of an equation are defined and equal, or both sides are undefined. An L-program p and an N-program q are functionally equivalent if $\forall \mathtt{d} \in D$: $[\![\mathtt{p}]\!]_{\mathtt{L}}\,\mathtt{d} = [\![\mathtt{q}]\!]_{\mathtt{N}}\,\mathtt{d}$. Wherever there is no danger of confusion, we omit the language index and assume that a language L is intended.

We briefly define the notions of interpreter and compiler.

Definition 1 (interpreter). *An* L*-program* int *is an* N/L-interpreter *iff* $\forall \mathtt{p} \in P_{\mathtt{N}}$ *and* $\forall \mathtt{d} \in D$*:*

$$[\![\mathtt{int}]\!]_{\mathtt{L}}(\mathtt{p},\mathtt{d}) \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{N}}\,\mathtt{d}. \tag{1}$$

Definition 2 (compiler). *An* L*-program* comp *is an* N-to-L-compiler *iff* $\forall \mathtt{p} \in P_{\mathtt{N}}$ *and* $\forall \mathtt{d} \in D$*:*

$$[\![[\![\mathtt{comp}]\!]_{\mathtt{L}}\,\mathtt{p}]\!]_{\mathtt{L}}\,\mathtt{d} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{N}}\,\mathtt{d}. \tag{2}$$

2.2 Specialization of Programs

We shall assume that all programs that we specialize have two parameters, that the first is always *static* (known) and that the second is always *dynamic* (unknown). In addition, we omit the binding-time division ("SD"). The definitions can be easily generalized to programs with more than two parameters and different orders of static and dynamic parameters at the cost of a more complex notation. Programs with multiple parameters can always be transformed such that they have two parameters.

Definition 3 (specializer). *An* L*-program* s *is an* L-specializer *iff* $\forall \mathtt{p} \in P_{\mathtt{L}}$ *and* $\forall \mathtt{x}, \mathtt{y} \in D$*:*

$$[\![[\![\mathtt{s}]\!]_{\mathtt{L}}(\mathtt{p},\mathtt{x})]\!]_{\mathtt{L}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{L}}(\mathtt{x},\mathtt{y}). \tag{3}$$

Specializing a *subject program* p with respect to static data x yields a *residual program*, $\mathtt{r} = [\![\mathtt{s}]\!]_{\mathtt{L}}(\mathtt{p},\mathtt{x})$. Practical specializers are discussed by Jones *et al*. [28].

Examples of specializers that have been designed and implemented include Mix [29], Schism [8], Similix [5] and Unmix [39] for functional languages; Ecce [35] and Logimix [38] for logic languages; C-mix [1], FCL-mix [24] and FSpec [32] for imperative languages; and the program transformers GPC [17] and SCP [47].

3. The Three Futamura Projections Revisited

We begin by a brief review of the construction of the three Futamura projections [13, 14] and draw the reader's attention to the abstraction scheme behind the projections. This scheme later plays an important role in this paper. Readers familiar with partial evaluation may still wish to review this section.

3.1 Abstraction

We begin with the initial expression,

$$[\![\mathtt{p}]\!]\,(\mathtt{x},\mathtt{y}) \quad = \quad \mathtt{out}, \tag{4}$$

#	1st abstract	2nd instantiate
	$[\![\mathtt{p}]\!]\,(\mathtt{x},\ \mathtt{y}) = \mathtt{out}$	
1.	$[\![\mathtt{s}]\!]\,(\mathtt{p},\ \mathtt{x}) = \mathtt{res}$	$[\![\mathtt{res}]\!]\,\mathtt{y} = \mathtt{out}$
2.	$[\![\mathtt{s}]\!]\,(\mathtt{s},\ \mathtt{p}) = \mathtt{gen}$	$[\![\mathtt{gen}]\!]\,\mathtt{x} = \mathtt{res}$
3.	$[\![\mathtt{s}]\!]\,(\mathtt{s},\ \mathtt{s}) = \mathtt{cog}$	$[\![\mathtt{cog}]\!]\,\mathtt{p} = \mathtt{gen}$

Figure 1. The three Futamura projections: equation system

which describes the application of a program p to two arguments x and y. By applying a specializer s to program p and data x, we abstract from the second argument y in this expression and obtain a *residual program* res that depends only on the second argument:

$$\mathit{let}\ \ [\![\mathtt{s}]\!]\,(\mathtt{p},\mathtt{x}) = \mathtt{res}\ \ \mathit{in}\ \ [\![\mathtt{res}]\!]\,\mathtt{y} = \mathtt{out}. \tag{5}$$

The correctness of computing p in two stages follows immediately from the correctness of s. We obtain the characteristic equation of this first abstraction step by combining expressions (4) and (5):

$$[\![\mathtt{res}]\!]\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]\,(\mathtt{x},\mathtt{y}). \tag{6}$$

The process of abstraction can be continued by taking $[\![\mathtt{s}]\!]\,(\mathtt{p},\mathtt{x})$ in (5) as the initial expression and applying s to s and p. The result is a *generating extension* gen *of* p that computes the same residual program res when applied to x as the specializer [11]:

$$\mathit{let}\ \ [\![\mathtt{s}]\!]\,(\mathtt{s},\mathtt{p}) = \mathtt{gen}\ \ \mathit{in}\ \ [\![\mathtt{gen}]\!]\,\mathtt{x} = \mathtt{res}. \tag{7}$$

We obtain the characteristic equation of the second abstraction step:

$$[\![[\![\mathtt{gen}]\!]\,\mathtt{x}]\!]\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]\,(\mathtt{x},\mathtt{y}). \tag{8}$$

After a third and last abstraction step that takes $[\![\mathtt{s}]\!]\,(\mathtt{s},\mathtt{p})$ in (7) as the initial expression, we arrive at the third projection, which yields a compiler generator cog that, when applied to p, produces p's generating extension gen:

$$\mathit{let}\ \ [\![\mathtt{s}]\!]\,(\mathtt{s},\mathtt{s}) = \mathtt{cog}\ \ \mathit{in}\ \ [\![\mathtt{cog}]\!]\,\mathtt{p} = \mathtt{gen}. \tag{9}$$

By composing the equations, we obtain the equation characteristic of a *compiler generator*.[1] Because of the importance of compiler generators later in this paper, we provide a separate definition.

Definition 4 (compiler generator). *An* L*-program* cog *is an* L-compiler generator *iff* $\forall \mathtt{p} \in P_{\mathtt{L}}$ *and* $\forall \mathtt{x}, \mathtt{y} \in D$*:*

$$[\![[\![[\![\mathtt{cog}]\!]_{\mathtt{L}}\,\mathtt{p}]\!]_{\mathtt{L}}\,\mathtt{x}]\!]_{\mathtt{L}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{L}}(\mathtt{x},\mathtt{y}). \tag{10}$$

The complete equation system that we obtain by repeating the abstraction scheme three times is summarized in Fig. 1, where the three *Futamura projections* in the left column are numbered (1., 2., and 3.) and the instantiations with the second argument can be seen in the right column. This equation system is the result of stepwise abstracting all three components (p, x, y) from the initial expression (4). The second and third Futamura projections involve *self-application* of a specializer, using s to specialize a copy of itself, and involve single and double self-applications, respectively.

Practical specializers that can perform all three Futamura projections and produce non-trivial generating extensions and compiler generators have been built for programming languages such as Scheme, Prolog, and C. Based on this practical experience, the modern approach of directly writing compiler generators was developed: *i.e*. hand-writing cog, instead of writing self-applicable specializers and generating the compiler generators according to

[1] For historical reasons and because the term 'compiler generator' is widely known in connection with the Futamura projections, we prefer to use it rather than 'program-generator generator' [44]. The former term originates from the original formulation of the projections, where p is an interpreter.

the third Futamura projection. The theoretical possibility of producing generating extensions and compiler generators by self-application of specializers was discovered independently by Futamura [13, 14] and by Turchin [49, 46], and brought to wider attention by Ershov [11, 12].

3.2 Instantiation

The residual program `res` in (5) is fixed with respect to `x`, but is independent of `y`. Instead of applying the residual program only to `y`, we can apply it to many different arguments (y_0, y_1, y_2, ...):

$$[\![\texttt{res}]\!]\,y_0 = \texttt{out}_0 \quad (11)$$
$$[\![\texttt{res}]\!]\,y_1 = \texttt{out}_1 \quad (12)$$
$$[\![\texttt{res}]\!]\,y_2 = \texttt{out}_2 \quad (13)$$
$$\ldots \qquad \ldots$$

An obvious advantage is computation speed. Whenever a program `p` is applied to many different argument pairs `(x,y)`, where `x` changes less frequently than `y` and a significant part of the computation of `p` depends on `x`, then it usually pays to abstract from `y` in the computation and to precompute `p` with `x` using a specializer. Many practical examples are discussed in Jones *et al*. [28], including computer graphics, language parsing and compilation.

3.3 Transition Scheme

Abstracting from the second argument, while fixing the first argument by a specializer, is the scheme underlying the construction of the three Futamura projections. This scheme can be applied mechanically and to any initial expression of the form given in (4). We summarize it as follows.

1. *Let* $[\![\texttt{p}]\!](\texttt{x},\texttt{y})$ *be an initial expression, where* `p` *is a program and* `(x,y)` *is a pair of arguments.*
2. *Abstract from the second argument by applying a specializer* `s`*, such that* $[\![\texttt{s}]\!](\texttt{p},\texttt{x}) = \texttt{res}$*. The residual program* `res` *is then parameterized with respect to the second argument.*

This scheme of abstraction and transition to a new computation can be applied repeatedly. There is no given boundary to the number of repetitions. In the Futamura projections it was applied three times. The scheme uses a fixed abstraction pattern (second argument) and a particular meta-program (specializer), and is an instance of the general class of Turchin's *metasystem transition schemes* [48].

3.4 Staging Computations as a General Principle

The abstraction scheme used in the Futamura projections is important because it covers the staging of a surprisingly large class of practical applications, including the transformation of interpreters into compilers and universal parsers into language parsers. Ershov coined the term *generating extension* to describe the entire class of program generators capable of generating residual programs [11].

> "*All processes which are more or less directly connected with an adaptation of universal components to predetermined parameters (concrete grammar, hardware parameters, environment inquiries, problem dimension, volume of data, etc) can be implemented with a uniform technique based on the mixed computation procedure.*" [11, p. 41]

The universality of this principle can be illustrated by a few two-argument programs and their generating extensions: interpreters and compilers, string matchers and matcher generators, and universal parsers and parser generators.

program	generating extension
$[\![\texttt{int}]\!](\texttt{p},\texttt{d})$	$[\![[\![\texttt{comp}]\!]\,\texttt{p}]\!]\,\texttt{d}$
$[\![\texttt{match}]\!](\texttt{pat},\texttt{txt})$	$[\![[\![\texttt{matchgen}]\!]\,\texttt{pat}]\!]\,\texttt{txt}$
$[\![\texttt{parse}]\!](\texttt{grm},\texttt{txt})$	$[\![[\![\texttt{parsegen}]\!]\,\texttt{grm}]\!]\,\texttt{txt}$

This short list indicates the fundamental importance of the Futamura projections for the theory of program generation. It also indicates that compiler generators, which can turn programs into their generating extensions, are powerful, universal tools for software development.

Example. We conclude our brief appraisal with the classic interpreter-compiler example of program specialization. Let `p` be an `N`-program, let `int` be an `N`/`L`-interpreter, and let `s` be an `L`-specializer. Take the initial expression $[\![\texttt{int}]\!]_{\texttt{L}}(\texttt{p},\texttt{d})$. Then the first Futamura projection,

$$\textit{let}\ \ [\![\texttt{s}]\!]_{\texttt{L}}(\texttt{int},\texttt{p}) = \texttt{q}\ \ \textit{in}\ \ [\![\texttt{q}]\!]_{\texttt{L}}\,\texttt{d} = [\![\texttt{p}]\!]_{\texttt{N}}\,\texttt{d}, \quad (14)$$

achieves `N`-to-`L`-compilation by specializing the `N`/`L`-interpreter. We see that `p` is an `N`-program, while `q` is an `L`-program (recall the definition of an interpreter (Def. 1) with $[\![\texttt{int}]\!]_{\texttt{L}}(\texttt{p},\texttt{d}) = [\![\texttt{p}]\!]_{\texttt{N}}\,\texttt{d}$).

The compiler generator `cog` produced by the third Futamura projection converts `int` into an `N`-to-`L`-compiler `comp`:

$$\textit{let}\ \ [\![\texttt{cog}]\!]_{\texttt{L}}\,\texttt{int} = \texttt{comp}\ \ \textit{in}\ \ [\![\texttt{comp}]\!]_{\texttt{L}}\,\texttt{p} = \texttt{q}. \quad (15)$$

This theoretical insight has important practical applications. Interpreters can be converted into compilers that compete with commercial and optimizing compilers for various programming languages and domain-specific languages (*e.g.*, [31, 43]).

4. Self-Generation of Compiler Generators

Futamura found that a compiler generator produced by the third Futamura projection can generate a copy of itself given a specializer [15]. In this section we study this curious feature in more detail.

4.1 Abstraction

Let us use the third Futamura projection (Fig. 1) as the initial expression and apply the abstraction scheme once more. For clarity, we underline the second argument ($\underline{\texttt{s}}$) in the initial expression:

#	1st abstract	2nd instantiate
3.	$[\![\texttt{s}]\!](\texttt{s},\ \underline{\texttt{s}}) = \texttt{cog}$	
4.	$[\![\texttt{s}]\!](\texttt{s},\ \texttt{s}) = \texttt{cog}$	$[\![\texttt{cog}]\!]\,\underline{\texttt{s}} = \texttt{cog}$

It follows from the staging of the initial expression (3.) that its result (`cog`) can also be obtained by applying `cog`, the program generated by the last projection (4.), to $\underline{\texttt{s}}$. The correctness of `s` applied in the last projection to `s` and `s` demands that both applications produce the same compiler generator:

Self-generation of a compiler generator:

$$\boxed{[\![\texttt{cog}]\!]\,\underline{\texttt{s}} = \texttt{cog} = [\![\texttt{s}]\!](\texttt{s},\underline{\texttt{s}})} \quad (16)$$

We see that the compiler generator `cog` generates a copy of itself. Such a compiler generator is called self-generating.

Definition 5 (self-generation). *A compiler generator* `cog` *is* self-generating *iff there is a specializer* `s` *such that* $[\![\texttt{cog}]\!]\,\texttt{s} = \texttt{cog}$.

While the two projections (3., 4.) are identical and, thus, generate the same program (`cog`), they are different in that the fourth projection is an abstraction of the third projection. They are identical only because the *abstracted specializer* ($\underline{\texttt{s}}$) and the *abstracting specializer* (`s`) happen to be identical. We quote Futamura regarding the existence of the fourth projection:

> "*If you substitute* α *for* int *in the third projection, you obtain* $\alpha(\alpha,\alpha)(\alpha) = \alpha(\alpha,\alpha)$ [15]. *You may call this the fourth projection. At DIKU, where* α *is called "mix" after Ershov, I hear that the fourth projection is nicknamed "the mixpoint" (the nickname is actually due to Peter Sestoft). The existence of a fifth projection is hardly imaginable since there is nothing left to specialize with respect to.*" [16, p. 378]

Thus, we have good reasons to call projection 4 the *fourth Futamura projection*. The first self-generation of a compiler generator was reported for the offline partial evaluator Mix [29]. Self-generation is sometimes used as a test for self-applicable specializers. Although any compiler generator produced by the third Futamura projection is self-generating by construction (*cf.* Thm. 1 below), it is nevertheless reassuring that a speedup is achieved in practice by using `cog` to produce `cog` instead of using the third Futamura projection (*e.g.*, [4, 24, 25, 30, 37, 38]).

One difference between *self-generating* and *self-printing* programs is that a compiler generator does not self-generate for all arguments, only for a certain argument, which must be a specializer.[2] In addition, a trivial self-generating compiler generator can always be obtained from a trivial specializer by double self-application, which also proves the existence of self-generating compiler generators. It is still astonishing in practice, although theoretically easy to prove, that rather large and sophisticated compiler generators produced by the third Futamura projection do self-generate (*e.g.*, the compiler generators of Similix [5] and C-mix [1]).

The following property relates self-generating compiler generators and those produced by the third Futamura projection.

Theorem 1. *The class of self-generating compiler generators is equal to the class of compiler generators produced by the third Futamura projection.*

Proof. Every self-generating compiler generator is also produced by the third Futamura projection. Let `s` be a specializer for which `cog` is self-generating: $[\![\texttt{cog}]\!]\,\texttt{s} = \texttt{cog}$. The following equalities hold (by instantiating the characteristic equation (Def. 4) with `s` in (17) and using the self-generation assumption to obtain (18-20)):

$$\begin{aligned}
[\![\texttt{s}]\!](\texttt{s},\texttt{s}) &= [\![[\![[\![\texttt{cog}]\!]\,\texttt{s}]\!]\,\texttt{s}]\!]\,\texttt{s} && (17)\\
&= [\![[\![\texttt{cog}]\!]\,\texttt{s}]\!]\,\texttt{s} && (18)\\
&= [\![\texttt{cog}]\!]\,\texttt{s} && (19)\\
&= \texttt{cog}. && (20)
\end{aligned}$$

The other direction: every compiler generator produced by the third Futamura projection is self-generating. For every specializer `s` (by using Def. 3 in (22) and equality (21) in (23)):

$$\begin{aligned}
\texttt{cog} &= [\![\texttt{s}]\!](\texttt{s},\texttt{s}) && (21)\\
&= [\![[\![\texttt{s}]\!](\texttt{s},\texttt{s})]\!]\,\texttt{s} && (22)\\
&= [\![\texttt{cog}]\!]\,\texttt{s}. && (23)
\end{aligned}$$

□

We see that self-generation is a property independent of the power of the specializer `s` used to generate the compiler generator `cog`; it is a structural property of the third Futamura projection. Also, the theorem indicates that the Futamura projections cannot be outsmarted by hand-writing a self-generating compiler generator that cannot be generated by the third Futamura projection. Thus, whenever a compiler generator is self-generating for some `s`, it can also be generated by double self-application of `s`.

[2] A program `p` is *self-printing* iff $\forall$`d`: $[\![\texttt{p}]\!]\,\texttt{d} = \texttt{p}$ [7]. No self-printing program can be a correct compiler generator. Self-printing programs and self-generating compilers generators are two disjoint program classes that represent two different forms of self-referential programs.

4.2 Instantiation

There is one more thing.

The fourth projection, $[\![\texttt{s}]\!](\texttt{s},\texttt{s}) = \texttt{cog}$, results from abstracting the second argument (`s`) from the third Futamura projection, $[\![\texttt{s}]\!](\texttt{s},\underline{\texttt{s}})$. Residual programs can be applied to many different arguments; in particular the residual program `cog` produced by the fourth projection can be applied to many different specializers ($\texttt{s}_0$, $\texttt{s}_1$, $\texttt{s}_2$, ...), not only to `s`, and each specializer will be turned into a compiler generator ($\texttt{cog}_0$, $\texttt{cog}_1$, $\texttt{cog}_2$, ...):

$$\begin{aligned}
[\![\texttt{cog}]\!]\,\texttt{s}_0 &= \texttt{cog}_0 && (24)\\
[\![\texttt{cog}]\!]\,\texttt{s}_1 &= \texttt{cog}_1 && (25)\\
[\![\texttt{cog}]\!]\,\texttt{s}_2 &= \texttt{cog}_2 && (26)\\
\cdots &\quad \cdots
\end{aligned}$$

Thus, there are very good reasons to identify the projection as the *fourth Futamura projection*. This projection predicts that specializers can be turned into compiler generators without self-applying the specializers. In addition, self-generation (16) is a *special case* of applying the compiler generator to a specializer. In general, the applications ((24), (25),...) do not self-generate. At first glance, the fourth Futamura projection in Sect. 4.1 appears disappointing, since it seems to produce nothing new. However, it is useful when applying the compiler generator to different specializers.

The properties of these compiler generators are analyzed in more detail in the following section. For now we note that a compiler generator turns programs into generating extensions; in particular, a specializer is turned into a compiler generator. Hence, Ershov's collection of generating extensions (Sect. 3.4) can be extended with one more interesting application: *a compiler generator is a generating extension of a specializer*.

program	generating extension
$[\![\texttt{s}]\!](\texttt{p},\texttt{x})$	$[\![[\![\texttt{cog}]\!]\,\texttt{p}]\!]\,\texttt{x}$

4.3 Application: Quasi-Online Compiler Generator

Suppose that we have an online specializer, $\texttt{s}_{\texttt{on}}$, and an offline compiler generator, $\texttt{cog}_{\texttt{off}}$, and that we wish to produce generating extensions that have the residual-program generation power of the online specializer (*e.g.*, [17, 40, 47]). Unfortunately, the online specializers constructed so far are not fully self-applicable and, hence, no online compiler generator is available to produce the desired generating extensions.

A solution to our generation problem is *quasi self-application* [18], which applies an offline specializer to our online specializer:

$$\texttt{gen}_{\texttt{off,on}} = [\![\texttt{s}_{\texttt{off}}]\!](\texttt{s}_{\texttt{on}},\texttt{p}). \qquad (27)$$

The generating extension $\texttt{gen}_{\texttt{off,on}}$ produced here is implemented by the offline specializer $\texttt{s}_{\texttt{off}}$, thus running with 'offline' efficiency, but has the residual-program generation power of the online specializer $\texttt{s}_{\texttt{on}}$. Thus, the residual programs generated by $\texttt{gen}_{\texttt{off,on}}$ run with 'online' efficiency. However, in our scenario we have only a (say, commercial) $\texttt{cog}_{\texttt{off}}$ and not the source code $\texttt{s}_{\texttt{off}}$ from which $\texttt{cog}_{\texttt{off}}$ was generated.

The generation problem can now be solved in a new way: first, $\texttt{cog}_{\texttt{off}}$ can be applied to $\texttt{s}_{\texttt{on}}$ to produce a new compiler generator:

$$\texttt{cog}_{\texttt{off,on}} = [\![\texttt{cog}_{\texttt{off}}]\!]\,\texttt{s}_{\texttt{on}}. \qquad (28)$$

Then, program `p` can be turned into the desired generating extension using the new compiler generator:

$$\texttt{gen}_{\texttt{off,on}} = [\![\texttt{cog}_{\texttt{off,on}}]\!]\,\texttt{p}. \qquad (29)$$

Thus, using an offline compiler generator, we can perform the analogue of quasi self-application (27) and produce a *quasi-online compiler generator* (28). One step is sufficient to turn $\mathtt{s}_{\mathtt{on}}$, without self-applying $\mathtt{s}_{\mathtt{on}}$, into a compiler generator. For example, $\mathtt{s}_{\mathtt{on}}$ may be an online specializer that we prototyped for a domain-specific language and $\mathtt{cog}_{\mathtt{off}}$ may be an advanced offline compiler generator. The languages involved in a compiler generator will be examined in Sect 7.1. Clearly, the implementation language of $\mathtt{s}_{\mathtt{on}}$ and the subject language of $\mathtt{cog}_{\mathtt{off}}$ must be identical.

That the two steps (28, 29) are a correct solution to our generation problem (27) can be seen from the following equalities.

$$[\![[\![\mathtt{cog}_{\mathtt{off}}]\!]\,\mathtt{s}_{\mathtt{on}}]\!]\,\mathtt{p} = [\![[\![[\![\mathtt{s}_{\mathtt{off}}]\!](\mathtt{s}_{\mathtt{off}},\mathtt{s}_{\mathtt{off}})]\!]\,\mathtt{s}_{\mathtt{on}}]\!]\,\mathtt{p} \quad (30)$$

$$= [\![[\![\mathtt{s}_{\mathtt{off}}]\!](\mathtt{s}_{\mathtt{off}},\mathtt{s}_{\mathtt{on}})]\!]\,\mathtt{p} \quad (31)$$

$$= [\![\mathtt{s}_{\mathtt{off}}]\!](\mathtt{s}_{\mathtt{on}},\mathtt{p}). \quad (32)$$

We assumed that $\mathtt{cog}_{\mathtt{off}}$ was produced by self-application of some, not necessarily known, $\mathtt{s}_{\mathtt{off}}$. In general, any (hand-written) compiler generator can turn a specializer into a new compiler generator.

5. Fourth Futamura Projection: Keep Going!

We have seen that the process of abstraction can be continued beyond the three Futamura projections. But it does not stop at the fourth Futamura projection. In this section we continue the abstraction process until the sixth projection, where we obtain an equation system that is closed in some sense. This important insight is due to Klimov and Romanenko [33, p. 6-7], but has not been studied in more depth. We shall do so in the following sections.

5.1 Abstraction

In the previous section we started with the third Futamura projection as the initial expression, and performed one step of abstraction. We now repeat the abstraction process three times until we arrive at the following equation system. To visualize how the abstraction proceeds stepwise, we underline the three specializers in the initial expression:

#	1st abstract		2nd instantiate
	$[\![\underline{\mathtt{s}}]\!](\underline{\mathtt{s}},\ \underline{\mathtt{s}})$	$=\mathtt{cog}$	
4.	$[\![\mathtt{s}]\!](\underline{\mathtt{s}},\ \underline{\mathtt{s}})$	$=\mathtt{cog}$	$[\![\mathtt{cog}]\!]\,\underline{\mathtt{s}} = \mathtt{cog}$
5.	$[\![\mathtt{s}]\!](\mathtt{s},\ \underline{\mathtt{s}})$	$=\mathtt{cog}$	$[\![\mathtt{cog}]\!]\,\underline{\mathtt{s}} = \mathtt{cog}$
6.	$[\![\mathtt{s}]\!](\mathtt{s},\ \mathtt{s})$	$=\mathtt{cog}$	$[\![\mathtt{cog}]\!]\,\underline{\mathtt{s}} = \mathtt{cog}$

The three projections (4., 5., 6.) in the left column are identical and yield the same program cog. Again, it appears as if no progress has been made. However, careful investigation shows that only at the last projection (6.) are all three components of the initial expression abstracted ($\underline{\mathtt{s}}$, $\underline{\mathtt{s}}$, $\underline{\mathtt{s}}$). Although the three projections are identical, the abstraction scheme has been performed three times. Going into reverse and instantiating the program cog produced by the sixth projection (in the last row) with $\underline{\mathtt{s}}$ rebuilds the compiler generator cog produced by the fifth projection, and so forth. Composing the equations in the right column, we obtain the characteristic equation:

Triple self-generation:

$$\boxed{[\![[\![[\![\mathtt{cog}]\!]\,\underline{\mathtt{s}}]\!]\,\underline{\mathtt{s}}]\!]\,\underline{\mathtt{s}} \;=\; \mathtt{cog} \;=\; [\![\underline{\mathtt{s}}]\!](\underline{\mathtt{s}},\underline{\mathtt{s}})} \quad (33)$$

We are not better off than before. Every cog in the generation series on the left-hand side is used at its *self-generation point* $\underline{\mathtt{s}}$. The three successive self-generations are of little practical interest. The projections (4., 5., 6.) are identical because the abstracted and the abstracting specializer are identical at each step of their construction. This, however, is due to the way we chose to construct the three projections. (In this case, equation (33) can also be obtained from the characteristic equation (Def. 4) by replacing each p, x, y by $\underline{\mathtt{s}}$.)

#	1st abstract		2nd instantiate
	$[\![\mathtt{s}_0]\!](\mathtt{s}_0,\ \mathtt{s}_0)$	$=\mathtt{cog}_{000}$	
4.	$[\![\mathtt{s}_0]\!](\mathtt{s}_0,\ \mathtt{s}_0)$	$=\mathtt{cog}_{000}$	$[\![\mathtt{cog}_{011}]\!]\,\mathtt{s}_1 = \mathtt{cog}_{111}$
5.	$[\![\mathtt{s}_0]\!](\mathtt{s}_0,\ \mathtt{s}_0)$	$=\mathtt{cog}_{000}$	$[\![\mathtt{cog}_{001}]\!]\,\mathtt{s}_1 = \mathtt{cog}_{011}$
6.	$[\![\mathtt{s}_0]\!](\mathtt{s}_0,\ \mathtt{s}_0)$	$=\mathtt{cog}_{000}$	$[\![\mathtt{cog}_{000}]\!]\,\mathtt{s}_1 = \mathtt{cog}_{001}$

Figure 2. Futamura projections: keep going

5.2 Instantiation

To analyze the equation system (4., 5., 6.), we shall define a convenient notation using two specializers, $\mathtt{s}_0$ and $\mathtt{s}_1$, and use a naming convention for the compiler generators produced from these two specializers. The index of the compiler generator indicates in what way the two specializers were involved in its generation:

$$\mathtt{cog}_{000} = [\![\mathtt{s}_0]\!](\mathtt{s}_0,\mathtt{s}_0) \quad (34)$$

$$\mathtt{cog}_{001} = [\![\mathtt{s}_0]\!](\mathtt{s}_0,\mathtt{s}_1) \quad (35)$$

$$\mathtt{cog}_{011} = [\![\mathtt{s}_0]\!](\mathtt{s}_1,\mathtt{s}_1) \quad (36)$$

$$\mathtt{cog}_{111} = [\![\mathtt{s}_1]\!](\mathtt{s}_1,\mathtt{s}_1) \quad (37)$$

We examine the application of the four compiler generators to $\mathtt{s}_1$:

$$[\![\mathtt{cog}_{000}]\!]\,\mathtt{s}_1 = [\![[\![\mathtt{s}_0]\!](\mathtt{s}_0,\mathtt{s}_0)]\!]\,\mathtt{s}_1 = [\![\mathtt{s}_0]\!](\mathtt{s}_0,\mathtt{s}_1) = \mathtt{cog}_{001} \quad (38)$$

$$[\![\mathtt{cog}_{001}]\!]\,\mathtt{s}_1 = [\![[\![\mathtt{s}_0]\!](\mathtt{s}_0,\mathtt{s}_1)]\!]\,\mathtt{s}_1 = [\![\mathtt{s}_0]\!](\mathtt{s}_1,\mathtt{s}_1) = \mathtt{cog}_{011} \quad (39)$$

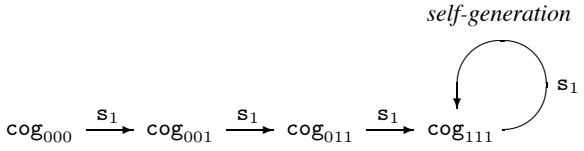
$$[\![\mathtt{cog}_{011}]\!]\,\mathtt{s}_1 = [\![[\![\mathtt{s}_0]\!](\mathtt{s}_1,\mathtt{s}_1)]\!]\,\mathtt{s}_1 = [\![\mathtt{s}_1]\!](\mathtt{s}_1,\mathtt{s}_1) = \mathtt{cog}_{111} \quad (40)$$

$$[\![\mathtt{cog}_{111}]\!]\,\mathtt{s}_1 = [\![[\![\mathtt{s}_1]\!](\mathtt{s}_1,\mathtt{s}_1)]\!]\,\mathtt{s}_1 = [\![\mathtt{s}_1]\!](\mathtt{s}_1,\mathtt{s}_1) = \mathtt{cog}_{111} \quad (41)$$

The first column shows the applications to $\mathtt{s}_1$. The second column replaces each compiler generator by its definition. The third column simplifies the expressions using Def. 3 for $\mathtt{s}_0$ and $\mathtt{s}_1$, respectively, and the fourth column states the results using the above definitions.

The results show that self-generation occurs only in the last case (41), while other compiler generators are produced in the first three cases (38-40). It is also apparent that $\mathtt{cog}_{011}$ and $\mathtt{cog}_{111}$ are two *functionally equivalent* compiler generators, although they may be quite different textually, because they are implemented by two different specializers, $\mathtt{s}_0$ and $\mathtt{s}_1$ (third and fourth columns of (39) and (40)). These two specializers may produce textually different residual programs when applied to the same arguments.

Looking at the equations syntactically, we observe that, in each step, the subscripts of the compiler generators are shifted to the left by one position. The following graph illustrates the series of compiler generators generated by (38-41):



We use these properties in Fig. 2, where $\mathtt{s}_0$ is used for the three abstraction steps that lead from the initial expression to the last projection (6.) and $\mathtt{s}_1$ is the argument in the instantiation steps that return us from (6.) to (4.). We arrive at an important relationship between the third Futamura projection and three successive compiler generator applications:

Bootstrapping a compiler generator:

$$\boxed{[\![[\![[\![\mathtt{cog}_{000}]\!]\,\mathtt{s}_1]\!]\,\mathtt{s}_1]\!]\,\mathtt{s}_1 \;=\; \mathtt{cog}_{111} \;=\; [\![\mathtt{s}_1]\!](\mathtt{s}_1,\mathtt{s}_1)} \quad (42)$$

Instead of using a double self-application of $\mathtt{s}_1$ in the third Futamura projection, the *same* compiler generator $\mathtt{cog}_{111}$ is produced

Futamura projection				Equivalent application
1.	$[\![\mathtt{s}]\!](\mathtt{p},\mathtt{x})$	$=$	$\mathtt{res}$	$=$ $[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{p}]\!]\,\mathtt{x}$
2.	$[\![\mathtt{s}]\!](\mathtt{s},\mathtt{p})$	$=$	$\mathtt{gen}$	$=$ $[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{s}]\!]\,\mathtt{p}$
3.	$[\![\mathtt{s}]\!](\mathtt{s},\mathtt{s})$	$=$	$\mathtt{cog}$	$=$ $[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{s}]\!]\,\mathtt{s}$

Figure 3. Alternatives for the three Futamura projections

in three steps starting from $\mathtt{cog}_{000}$. This step-by-step process progresses towards the ultimate goal, the generation of $\mathtt{cog}_{111}$. At each step the version generated during the previous step is used as a tool to produce the next version of the compiler generator. We have good reasons to call this the *bootstrapping of a compiler generator* [33]. We note that the initial compiler generator need not necessarily be produced by self-application. Rather, it can be any (hand-written) compiler generator. We can also say that the left-hand side of (42) is a staging of the third Futamura projection on the right-hand side.

It is important to bear in mind that the three-step bootstrapping does *not* circumvent the challenge of writing self-applicable specializers, but it only delays the challenge. The equality between the results of the three-step bootstrapping and the third Futamura projection in (42) requires that, if the latter leads to an inefficient compiler generator, so does the former, and vice versa.

5.3 Application: Alternatives to the Futamura Projections

This remarkable process is an alternative to the third Futamura projection. It follows that the classical three Futamura projections have equivalences in terms of compiler-generator applications, as shown in Fig. 3, where $\mathtt{s}$ can be any specializer and $\mathtt{cog}'$ can be any compiler generator. In the special case that $\mathtt{s}_0$ and $\mathtt{cog}_{000}$ are used, where the former is the self-generation point of the latter, the expressions in the right column can be simplified to the usual application of a compiler generator. The new equations also follow from an appropriate instantiation of the characteristic equation of a compiler generator (Def. 4).

(a) Using the second Futamura projection as an example, the programs from Ershov's collection in Sect. 3.4 can be turned into their generating extensions in another way:

$[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{s}]\!]\,\mathtt{int}$	$=$	$\mathtt{comp}$
$[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{s}]\!]\,\mathtt{match}$	$=$	$\mathtt{matchgen}$
$[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}]\!]\,\mathtt{s}]\!]\,\mathtt{parse}$	$=$	$\mathtt{parsegen}$

(b) Using the bootstrapping alternative to the third Futamura projection to produce in three steps an online compiler generator from an online specializer $\mathtt{s}_{\mathtt{on}}$:

$$[\![[\![[\![\mathtt{cog}']\!]\,\mathtt{s}_{\mathtt{on}}]\!]\,\mathtt{s}_{\mathtt{on}}]\!]\,\mathtt{s}_{\mathtt{on}} \quad = \quad \mathtt{cog}_{\mathtt{on,on,on}}. \tag{43}$$

5.4 Bootstrapping versus Double Self-Application

The three-step bootstrapping of $\mathtt{cog}_{111}$ on the left-hand side of (42) is quite different from the familiar *recipe for self-application* [28], which suggests an approach to the third Futamura projection in a *bottom-up fashion* (1., 2., 3.) starting with the specialization of a self-interpreter. Bootstrapping progresses in a *top-down fashion* (6., 5., 4.). Already the first step produces a working compiler generator $\mathtt{cog}_{001}$, without self-application of $\mathtt{s}_1$. The compiler generator $\mathtt{cog}_{011}$ produced by the second step is already functionally equivalent to the desired $\mathtt{cog}_{111}$. If functionality is what matters, we are done. We observe that the generation of $\mathtt{cog}_{011}$ involves the analogue of a single self-application, while the generation of $\mathtt{cog}_{111}$ by the third Futamura projection involves double self-application and that no compiler generator results until the task is fully successful. Bootstrapping a new compiler generator by starting from a mature compiler generator $\mathtt{cog}_{000}$ may thus have considerable practical advantages. To conclude, bootstrapping does not circumvent the problem of writing specializers that specialize well, but may give another perspective to this tricky task.

The time required to bootstrap a compiler generator is the *sum* of the times required for each individual step. The time required to generate a compiler generator by double self-application is the *product* of the computational overhead of each layer of self-application, which is similar to the use of several layers of interpreters, where each new level of interpretation may multiply the running time by a significant factor.

If we take the time required for the self-generation of a compiler generator as an indication of its efficiency when performing a bootstrapping step, then even three-step bootstrapping can be faster than double self-application. The self-generation speedup ratio reported for FCL-mix is about 8.3 [28, p. 97], indicating that three-step bootstrapping can be faster than double self-application (three-step bootstrapping pays off when the ratio is greater than 3). In contrast, the self-generation speedup ratio reported for Mix is only about 1.6 [30, p. 42], indicating that, for this specializer, double self-application is faster. Naturally, the choice also depends on the number of bootstrapping steps required. As discussed above, two steps are sufficient for applications where only the functionality of the compiler generator matters, whereas, for others, such as compiler generators for domain-specific languages, even a single step suffices (the latter will be discussed in Sect. 7.2).

Self-generation can be useful as a *partial test of correctness* during program development. Using the programs from Sect. 5.2: if the application of $\mathtt{cog}_{111}$ yields a program $\mathtt{cog}'_{111} = [\![\mathtt{cog}_{111}]\!]\,\mathtt{s}_1$ that is textually different from program $\mathtt{cog}_{111} = [\![\mathtt{cog}_{011}]\!]\,\mathtt{s}_1$, then $\mathtt{s}_1$ must contain an error, if we assume that the original program $\mathtt{s}_0$ (and $\mathtt{cog}_{000}$) is well-tested and most likely correct. The converse is not true: even if $\mathtt{cog}_{111} = \mathtt{cog}'_{111}$, there may still be errors in $\mathtt{s}_1$. The test is simple and can be performed mechanically. For example, it can be useful in an incremental development process where specializer $\mathtt{s}_1$ is an extension of a well-tested specializer $\mathtt{s}_0$. This is similar to partially validating a compiler by self-compilation [34].

6. Recursive Compiler Generation

We wish next to briefly consider the case of compiler generators that involves quasi self-application of more than two specializers:

$$\mathtt{cog}_{abc} \quad = \quad [\![\mathtt{s}_a]\!](\mathtt{s}_b, \mathtt{s}_c). \tag{44}$$

Compiler generator $\mathtt{cog}_{ijk}$ is produced by a three-step bootstrapping that consumes $\mathtt{s}_i$, $\mathtt{s}_j$, and $\mathtt{s}_k$,

$$\mathtt{cog}_{ijk} \quad = \quad [\![[\![[\![\mathtt{cog}_{abc}]\!]\,\mathtt{s}_i]\!]\,\mathtt{s}_j]\!]\,\mathtt{s}_k, \tag{45}$$

and $\mathtt{cog}_{abc}$ self-generates after three steps with $\mathtt{s}_a$, $\mathtt{s}_b$, and $\mathtt{s}_c$:

$$\mathtt{cog}_{abc} \quad = \quad [\![[\![[\![\mathtt{cog}_{abc}]\!]\,\mathtt{s}_a]\!]\,\mathtt{s}_b]\!]\,\mathtt{s}_c. \tag{46}$$

We observed the latter when the generation series came to a self-generation point after the same specializer was used three times (Sect. 5.2). In general, however, the generation series *progresses* by generating a new compiler generator as long as new specializers are supplied,

$$\mathtt{cog}_{ijk} \quad = \quad [\![[\![\ldots[\![[\![\mathtt{cog}_{abc}]\!]\,\mathtt{s}_d]\!]\ldots]\!]\mathtt{s}_i]\!]\mathtt{s}_j]\!]\,\mathtt{s}_k, \tag{47}$$

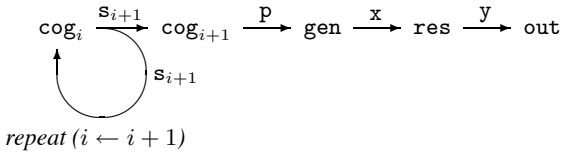
and *stagnates* by reproducing an old 'seen-before' compiler generator when the supply of specializers is finite:

$$\mathtt{cog}_{abc} \quad = \quad [\![[\![\ldots[\![[\![\mathtt{cog}_{abc}]\!]\,\mathtt{s}_d]\!]\ldots]\!]\mathtt{s}_a]\!]\mathtt{s}_b]\!]\,\mathtt{s}_c. \tag{48}$$

The quality of a compiler generator in terms of its operational and functional properties depends only on the *last three specializers*

involved in its generation. There is no "memory of the past" in this generation series beyond the last three generation steps. In this sense, and in the context of Futamura's abstraction scheme, we consider the projections as *closed after the sixth Futamura projection*. The generation is complete since any $\mathtt{cog}_{ijk}$ can be obtained from $\mathtt{cog}_{abc}$ by supplying $\mathtt{s}_i$, $\mathtt{s}_j$ and $\mathtt{s}_k$ in the appropriate order.

The graph below illustrates the recursive generation of compiler generators from an initial compiler generator $\mathtt{cog}_i$ by supplying a series of specializers $\mathtt{s}_{i+1}$ ($i = 0 \ldots$). The series stagnates and reproduces previously generated compiler generators when the supply of specializers is finite. It terminates when an 'ordinary' program p is supplied instead of a specializer. Self-generation is a stagnation of the recursive generation process that restarts to generate new compiler generators when new specializers are supplied. The process can be repeated any number of times.



7. An Application to Domain-Specific Languages and Language Extensions

For the sake of simplicity we restricted our analysis to the case of program generators that involve one language. There remains to discuss the case when more languages are involved. This can be applied to the generation of compiler generators for domain-specific languages and to the extension of the subject language of an existing compiler generator.

7.1 The Languages of Specializers and Compiler Generators

A specializer is characterized by *three languages*: the subject language A, the implementation language B, and the target language Z:

$$[\![[\![\mathtt{s}]\!]_{\mathtt{B}}(\mathtt{p},\mathtt{x})]\!]_{\mathtt{Z}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{A}}(\mathtt{x},\mathtt{y}). \tag{49}$$

A compiler generator is characterized by *four languages*: the subject language A, the implementation language X, the target language Y, and the target language Z of the generating extensions that it produces:

$$[\![[\![[\![\mathtt{cog}]\!]_{\mathtt{X}}\,\mathtt{p}]\!]_{\mathtt{Y}}\,\mathtt{x}]\!]_{\mathtt{Z}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{A}}(\mathtt{x},\mathtt{y}). \tag{50}$$

To express these properties, it is most convenient to use a variant of the T-diagrams familiar from compiler construction [10]. The T-diagram of a specializer is shown in Fig. 4. The bullet (•) in the center distinguishes it from a compiler. The T-diagram of a compiler generator is shown in Fig. 4, in which the target language of the generated generating extensions (Z) is written in the center.

Figure 5 defines the *rule of language inheritance* in terms of T-diagrams. When a compiler generator $\mathtt{cog}'$, characterized by

$$[\![[\![[\![\mathtt{cog}']\!]_{\mathtt{W}}\,\mathtt{p}]\!]_{\mathtt{X}}\,\mathtt{x}]\!]_{\mathtt{Y}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{B}}(\mathtt{x},\mathtt{y}), \tag{51}$$

is applied to the specializer s (49), then the compiler generator that this application generates has the same languages as cog (50).

By inspecting the diagrams, we find that language B, the implementation language of s and the subject language of $\mathtt{cog}'$, does not appear as one of the four languages characterizing cog. Similarly, the implementation language W of $\mathtt{cog}'$ does not appear in cog. Six languages characterize the original programs s and $\mathtt{cog}'$, four of which are inherited by cog. Note that Z, the target language of s, becomes the target language of the generating extensions that cog produces, that Y becomes the target language of cog and that X becomes its implementation language. This transformation changes the subject and target languages of the original compiler generator.

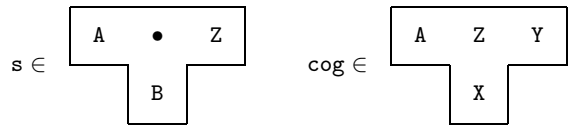


Figure 4. T-diagram: specializer s and compiler generator cog

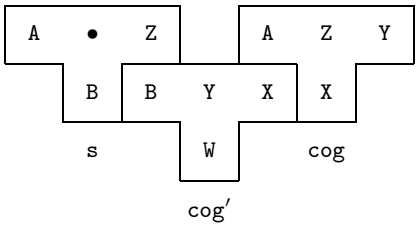


Figure 5. Language inheritance rule: $[\![\mathtt{cog}']\!]\,\mathtt{s} = \mathtt{cog}$

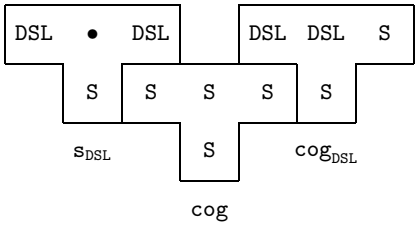


Figure 6. DSL compiler generator: $[\![\mathtt{cog}]\!]\,\mathtt{s}_{\mathtt{DSL}} = \mathtt{cog}_{\mathtt{DSL}}$

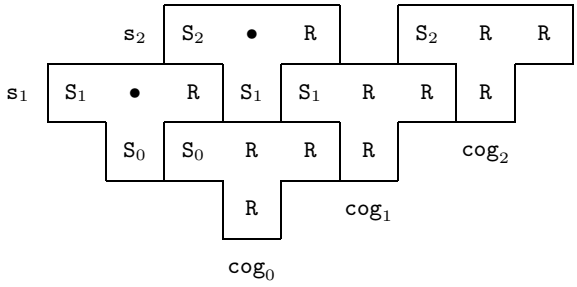


Figure 7. Subject language extension: $[\![[\![\mathtt{cog}_0]\!]\,\mathtt{s}_1]\!]\,\mathtt{s}_2 = \mathtt{cog}_2$

7.2 Application: Domain-Specific Languages

Domain-specific languages are important in the development of modern software. They allow programs to be more concise at a higher level of abstraction than conventional programming lan-

guages, but they continue to evolve with their application domain. It is thus important that tools supporting domain-specific program development be maintainable and implementable at low cost.

Domain-specific languages are usually not well suited for writing program generators. They may be too inefficient for complex program transformation algorithms due to their interpretive nature or other resources constraints (*e.g.*, embedded systems) or they may not be Turing-complete. Specializers have been used successfully to implement compilers for domain-specific languages (*e.g.*, [43]), but generating extensions or compiler generators have not been produced for these languages. Nevertheless, it is desirable to have available a variety of tools supporting domain-specific program development.

Suppose we want to convert DSL-programs into generating extensions that are implemented in a powerful symbol manipulation language, such as Scheme (S), but produce DSL-residual programs. Thus, these generating extensions can be run efficiently on a host machine S. Assume further that a compiler generator cog is available for S (*e.g.*, PGG [45]).

Now write a DSL-to-DSL-specializer $\mathtt{s}_{\mathtt{DSL}}$ in S that takes advantage of domain-specific knowledge for good DSL-specialization. It need not be self-applicable and can use online partial evaluation. Nevertheless, we can produce an efficient compiler generator that converts DSL-programs into the desired generating extensions by the following method. Given specializer $\mathtt{s}_{\mathtt{DSL}}$, characterized by

$$[\![[\![\mathtt{s}_{\mathtt{DSL}}]\!]_{\mathtt{S}}\,(\mathtt{p},\mathtt{x})]\!]_{\mathtt{DSL}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{DSL}}\,(\mathtt{x},\mathtt{y}), \tag{52}$$

and compiler generator cog,

$$[\![[\![[\![\mathtt{cog}]\!]_{\mathtt{S}}\,\mathtt{p}]\!]_{\mathtt{S}}\,\mathtt{x}]\!]_{\mathtt{S}}\,\mathtt{y} \quad = \quad [\![\mathtt{p}]\!]_{\mathtt{S}}\,(\mathtt{x},\mathtt{y}), \tag{53}$$

convert $\mathtt{s}_{\mathtt{DSL}}$ into a new compiler generator:

$$\mathtt{cog}_{\mathtt{DSL}} \quad = \quad [\![\mathtt{cog}]\!]_{\mathtt{S}}\,\mathtt{s}_{\mathtt{DSL}}. \tag{54}$$

The new compiler generator $\mathtt{cog}_{\mathtt{DSL}}$ turns DSL-programs into generating extensions that produce DSL-residual programs on machine S. Let p be a DSL-program, $[\![\mathtt{p}]\!]_{\mathtt{DSL}}\,(\mathtt{x},\mathtt{y})$. Then we obtain p's generating extension that cross-generates DSL-residual programs:

$$\mathtt{gen}_{\mathtt{DSL}} \quad = \quad [\![\mathtt{cog}_{\mathtt{DSL}}]\!]_{\mathtt{S}}\,\mathtt{p}, \tag{55}$$

$$\mathtt{res}_{\mathtt{DSL}} \quad = \quad [\![\mathtt{gen}_{\mathtt{DSL}}]\!]_{\mathtt{S}}\,\mathtt{x}. \tag{56}$$

After moving the residual program $\mathtt{res}_{\mathtt{DSL}}$ to the DSL-machine:

$$\mathtt{out} \quad = \quad [\![\mathtt{res}_{\mathtt{DSL}}]\!]_{\mathtt{DSL}}\,\mathtt{y}. \tag{57}$$

The languages involved in the conversion of $\mathtt{s}_{\mathtt{DSL}}$ into $\mathtt{cog}_{\mathtt{DSL}}$ by cog are shown in Fig. 6, which is an example of the language inheritance rule illustrated in Fig. 5.

One advantage of this method is that a domain expert, who may know how to specialize DSL-programs and how to define this in S, does not have to understand how to produce efficient generating extensions or how to achieve successful self-application. The main focus is on formulating good DSL specialization. This approach emphasizes the reuse of program-generation technology available in the form of a compiler generator. The domain expert can take advantage of the advanced features of a mature compiler generator. Moreover, staying within DSL when specializing programs may give better results than translating DSL-programs to S, generating S-residual programs with cog, and translating them back to DSL.

7.3 Application: Subject Language Extensions

Another form of bootstrapping builds up a compiler generator for a larger and larger subject language. Suppose we wish to extend the subject language $\mathtt{S}_0$ of a compiler generator $\mathtt{cog}_0$ to a larger one called $\mathtt{S}_1$, such that $P_{\mathtt{S}_0} \subseteq P_{\mathtt{S}_1}$. This extension can be made by defining the specialization of $\mathtt{S}_1$ in $\mathtt{S}_0$ and turning the new specializer $\mathtt{s}_1$ into a new compiler generator $\mathtt{cog}_1$ for $\mathtt{S}_1$ using $\mathtt{cog}_0$ itself as a tool (Fig. 7). Here, for simplicity, all target languages are R and not modified. For example, Amix is a specializer where the subject language is a high-level functional language and the target language is a low-level assembly language [26].

If the language extension is conservative, every $\mathtt{S}_0$-program will have the same semantics in $\mathtt{S}_1$. Self-generation can then serve as a partial correctness test; *i.e.*, there is an error in the new specializer $\mathtt{s}_1$ (or in the original $\mathtt{cog}_0$) if $\mathtt{cog}_1' \neq [\![\mathtt{cog}_1']\!]\,\mathtt{s}_1$, where $\mathtt{cog}_1' = [\![[\![[\![\mathtt{cog}_0]\!]\,\mathtt{s}_1]\!]\,\mathtt{s}_1]\!]\,\mathtt{s}_1$ (*cf.* Sect. 5.4).

Subsequently, successive language extensions ($\mathtt{S}_2$, $\mathtt{S}_3$, ...) can be implemented by successive expansion of $\mathtt{s}_1$ and by turning the new specializers ($\mathtt{s}_2$, $\mathtt{s}_3$, ...) into new compiler generators using the previously produced compiler generator as a transformation tool ($\mathtt{cog}_1$, $\mathtt{cog}_2$, ...). A new specializer for $\mathtt{S}_2$ need not be written in $\mathtt{S}_0$, but can be written in terms of $\mathtt{S}_1$, and so forth, thereby making use of new, more advanced language features. This is shown for two successive steps by combining the diagrams to a single diagram in Fig. 7. This process can be repeated as many times as necessary. Several useful applications may result from this incremental developmental process, where a first step may consist of writing a small compiler generator $\mathtt{cog}_0$ for a simple language $\mathtt{S}_0$.

$$\mathtt{cog}_1 \quad = \quad [\![\mathtt{cog}_0]\!]_{\mathtt{R}}\,\mathtt{s}_1 \tag{58}$$

$$\mathtt{cog}_2 \quad = \quad [\![\mathtt{cog}_1]\!]_{\mathtt{R}}\,\mathtt{s}_2 \tag{59}$$

$$\mathtt{cog}_3 \quad = \quad [\![\mathtt{cog}_2]\!]_{\mathtt{R}}\,\mathtt{s}_3 \tag{60}$$

$$\ldots \qquad\qquad \ldots$$

This process is similar to classical compiler bootstrapping [10, 34], but differs in that a specializer is turned into a 'more complex' compiler generator instead of translating a compiler into a compiler.

8. Related Work

Futamura was the first to observe theoretically the self-generation property of the compiler generators produced by his third projection [15]. The first actual self-generation was reported a few years later for a self-applicable offline partial evaluator [29] and has been used to assess the speed-up of self-applicable partial evaluators (*e.g.*, [5, 24, 25, 30, 37, 38]). Self-generation is usually regarded as a curiosity and not viewed as a special case of a more general use of compiler generators.

Klimov and Romanenko's important insight [33] was first described in English in publication [22], but has drawn little attention since. The abstraction scheme used in the Futamura projections, and the constructions in this paper are a special case of the more general metasystem transition schemes [46, 48, 19, 22]. Related applications are the generation of specializer from interpreters [18] and the unstaging of generating extensions by composition with self-interpreters [23].

Multi-level generating extensions [20] are a generalization of Ershov's two-level generating extensions that we studied in this paper. Multi-stage programming, which originates from the insights into hand-writing generating extensions and compiler generators, supports the development of type-safe program generators [42].

9. Conclusion

The modern approach of hand-writing compiler generators, while useful, deprives us of using the power and flexibility of various self-application schemes. In this paper we have proposed several non-standard applications of compiler generators, such as the generation of compiler generators for domain-specific languages, the generation of quasi-online compiler generators, and the bootstrapping of compiler generators as an alternative to the third Futamura projection. This may help in approaching some old program generation problems from a new angle, in particular problems involving

specializers that are not self-applicable and the production of generating extensions for domain-specific languages.

The approach proposed here may permit the generation of compiler generators for domain-specific languages, while avoiding the need of developing self-applicable specializers for these languages; the latter may be difficult when such languages are not geared towards symbol manipulation and program transformation tasks (Sect. 7.2).

Quasi-online compiler generators may provide a pragmatic approach toward combining the ability of generating compiler generators via (self-applicable) offline specializers with the ability to produce highly optimized residual programs via (non-self-applicable) online specializers (Sect. 4.3).

The bootstrapping of compiler generators may be an alternative to direct generation by the third Futamura projection (Sect. 5.3) and the extension of their subject languages may be useful for the development and evolution of compiler generators (Sect. 7.3). However, experiments in a realistic setting are needed to determine if and under what conditions this is practical.

At first, the steps beyond the third Futamura projection do not appear to make sense, as no further changes occur. But the situation is different when using different specializers in a mixed fashion. Since these specializers can have different characteristics, concerning, for example, specialization power, self-applicability and languages, several new application scenarios may emerge. In hindsight, it may not be surprising that Futamura's abstraction scheme can be repeated up to the sixth projection before the equation system is closed in some sense and that, as suggested by Futamura, the self-generation of compiler generators can indeed be connected to the fourth projection. Nevertheless, to our knowledge, except for the references cited, these foundational issues have not been studied in the literature.

While our aim was a more systematic and theoretical investigation of the potential and limitations of compiler generators, an experimental investigation of the applications proposed in this paper will be the next step, as it was for other applications proposed for the Futamura projections. Nevertheless, we believe that the theoretical ideas and insights into compiler generators and the classical Futamura projections are sufficiently interesting to warrant their discussion. Moreover, it may bring to a wider audience an answer to the question: What is beyond the third Futamura projection?

Acknowledgments

The author would like to thank the members of IFIP WG 2.11 (Program Generation), in particular, Olivier Danvy and Kevin Hammond, for constructive feedback about this work. Many thanks also to Andrei Klimov and Sergei Romanenko for many inspiring discussions. Thanks are also due to Kenichi Asai, Lars Hartmann, and the anonymous reviewers for their insightful comments. Part of this work was performed while the author was visiting the National Institute of Informatics (NII), Tokyo. It is a great pleasure to thank Akihiko Takano and Zhenjiang Hu for their hospitality and for providing excellent working conditions.

References

[1] L. O. Andersen. Partial evaluation of C and automatic compiler generation. In U. Kastens, P. Pfahler (eds.), *Compiler Construction. 4th International Conferences*, Lecture Notes in Computer Science, Vol. 641, 251–257. Springer-Verlag, 1992.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. Ph.D. thesis. DIKU Report 94/19, Dept. of Computer Science, University of Copenhagen, 1994.

[3] L. Birkedal, M. Welinder. Hand-writing program generator generators. In M. Hermenegildo, J. Penjam (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, Lecture Notes in Computer Science, Vol. 844, 198–214. Springer-Verlag, 1994.

[4] A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In J. Diaz, F. Orejas (eds.), *TAPSOFT'89*, Lecture Notes in Computer Science, Vol. 352, 81–96. Springer-Verlag, 1989.

[5] A. Bondorf, O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

[6] A. Bondorf, D. Dussart. Improving CPS-based partial evaluation: writing cogen by hand. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, 1–9. University of Melbourne, Australia, 1994.

[7] P. Bratley, J. Millo. Computer recreations: self-reproducing programs. *Software – Practice and Experience*, 2(4):397–400, 1972.

[8] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger (ed.), *ESOP'88*, Lecture Notes in Computer Science, Vol. 300, 236–247. Springer-Verlag, 1988.

[9] C. Consel, J. L. Lawall, A.-F. Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52(1-3):341–370, 2004.

[10] J. Earley, H. Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

[11] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.

[12] A. P. Ershov. On the essence of compilation. In E. Neuhold (ed.), *Formal Description of Programming Concepts*, 391–420. North-Holland, 1978.

[13] Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971. Reprinted in Higher-Order and Symbolic Computation, 12(4): 381–391, 1999.

[14] Y. Futamura. EL1 partial evaluator. Progress report, Center for Research in Computing Technology, Harvard University, 1973. Extract from the introduction reproduced in [16].

[15] Y. Futamura. Partial computation of programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, A. Yonezawa (eds.), *RIMS Symposia on Software Science and Engineering*, Lecture Notes in Computer Science, Vol. 147, 1–35. Springer-Verlag, 1983.

[16] Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[17] Y. Futamura, Z. Konishi, R. Glück. WSDFU: Program transformation system based on generalized partial computation. In T. Æ. Mogensen, D. Schmidt, I. H. Sudborough (eds.), *The Essence of Computation: Complexity, Analysis, Transformation*, Lecture Notes in Computer Science, Vol. 2566, 358–378. Springer-Verlag, 2002.

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

[19] R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti (ed.), *Logic Program Synthesis and Transformation. Proceedings*, Lecture Notes in Computer Science, Vol. 1048, 234–251. Springer-Verlag, 1996.

[20] R. Glück, J. Jørgensen. Efficient multi-level generating extensions for program specialization. In M. Hermenegildo, S. D. Swierstra (eds.), *Programming Languages: Implementations, Logics and Programs. Proceedings*, Lecture Notes in Computer Science, Vol. 982, 259–278. Springer-Verlag, 1995.

[21] R. Glück, J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

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

[23] R. Glück, A. V. Klimov. A regeneration scheme for generating extensions. *Information Processing Letters*, 62(3):127–134, 1997.

[24] C. K. Gomard, N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[25] C. K. Gomard, N. D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[26] C. K. Holst. Language triplets: the Amix approach. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 167–185. North-Holland, 1988.

[27] C. K. Holst, J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Fourth Annual Glasgow Workshop on Functional Programming. Draft Proceedings*, 210–218, 1991.

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

[29] N. D. Jones, P. Sestoft, H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 202, 124–140. Springer-Verlag, 1985.

[30] N. D. Jones, P. Sestoft, H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[31] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Conference Record of the Nineteenth Symposium on Principles of Programming Languages*, 258–268. ACM Press, 1992.

[32] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, R. Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.

[33] A. V. Klimov, S. A. Romanenko. Metavychislitel' dlja jazyka Refal. Osnovnye ponjatija i primery. (A metaevaluator for the language Refal. Basic concepts and examples). Preprint 71, Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1987. (in Russian).

[34] O. Lecarme, M.-C. Peyrolle-Thomas. Self-compiling compilers: an appraisal of their implementation and portability. *Software – Practice and Experience*, 8:149–170, 1978.

[35] M. Leuschel. The Ecce partial deduction system. In G. Puebla et al. (eds.), *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, Technical Report CLIP7/97.1. Polytechnic University of Madrid, Spain, 1997.

[36] M. Leuschel, J. Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator LOGEN. *Electronic Notes in Theoretical Computer Science*, 30(2):157–162, 2000.

[37] T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 39–44. ACM Press, 1995.

[38] T. Æ. Mogensen, A. Bondorf. Logimix: a self-applicable partial evaluator for Prolog. In K.-K. Lau, T. Clement (eds.), *Logic Program Synthesis and Transformation*, Workshops in Computing, 214–227. Springer-Verlag, 1993.

[39] S. A. Romanenko. The specializer Unmix, 1990. Program and documentation available from `ftp://ftp.diku.dk/pub/diku/dists/jones-book/Romanenko/`.

[40] M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[41] E. Sumii, N. Kobayashi. Online-and-offline partial evaluation: a mixed approach. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 12–21. ACM Press, 2000.

[42] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, M. Odersky (eds.), *Domain-specific program generation. Proceedings*, Vol. 3016, 30–50. Springer-Verlag, 2004.

[43] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[44] P. Thiemann. Cogen in six lines. In *ACM International Conference on Functional Programming*, 180–189. ACM Press, 1996.

[45] P. Thiemann. The PGG system: User manual, 2003. Program and documentation available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

[46] V. F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, 1979.

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

[48] V. F. Turchin. Metacomputation: metasystem transitions plus supercompilation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science, Vol. 1110, 481–509. Springer-Verlag, 1996.

[49] V. F. Turchin, And. V. Klimov, Ark. V. Klimov, V. F. Khoroshevsky, A. G. Krasovsky, S. A. Romanenko, I. B. Shchenkov, E. V. Travkina. *Bazisnyj Refal i ego realizacija na vychislitel'nykh mashinakh (Basic Refal and its implementation on computers)*. GOSSTROJ SSSR, CNIPIASS, 1977. (in Russian).