ELSEVIER

Information Processing Letters 62 (1997) 127-134

Information Processing Letters

A regeneration scheme for generating extensions [1]

Robert Glück [a,*], Andrei Klimov [b,2]

[a] DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

[b] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Miusskaya Square 4, RU-125047 Moscow, Russia

Received 8 November 1996; revised 26 February 1997
Communicated by H. Ganzinger

Abstract

A regeneration scheme is presented which shows how to change the computation staging of a generating extension by a two-level metasystem structure using program specialization and program composition. From the results described in this paper we can see that program generation and program degeneration are two extremes of the same transformation dimension; this relates several well-known program transformations under a general scheme. © 1997 Elsevier Science B.V.

Keywords: Programming languages; Program generation; Partial evaluation; Program composition; Metacomputation

1. Introduction

Generating extensions describe uniformly the functioning of various program generators. This has the big advantage that program generators can be implemented with uniform techniques, including diverse applications such as parsing, translation, graphics, and operating systems [1]. Generating extensions are a powerful concept because they capture the essence of apparently different program generators. Automatic tools for turning a general program into a generating extension now exist for various programming languages, such as Prolog, Scheme, and C (see [8]).

The division of a computation into two (or more) stages has been studied intensively in the area of *partial evaluation* [2,1,8]. However, a number of problems related to generating extensions has not been investigated systematically. This paper considers one of them: the direct conversion of generating extensions into generating extensions. In others words, the objective is to *turn a staged computation into another staged computation.*

Let us introduce the regeneration problem first. A *generating extension of a program P* is a program Gen_P that takes a part of P's input, and produces a specialized program that returns the same result when applied to the remaining input as program P when applied to all input. Suppose P is a program with four arguments and let $\langle P\ A,B,C,D\rangle$ denote the application of program P to its input A,B,C,D. Generating extension Gen_P is defined such that for all A,B,C,D:

* Corresponding author. Email: glueck@diku.dk.

[1] 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) 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.

[2] Email: and.klimov@refal.ac.msk.su.

$$\langle\langle Gen_P\ A, B\rangle\ C, D\rangle = \langle P\ A, B, C, D\rangle. \qquad (1)$$

The computation of P is performed in two stages: Gen_P takes A, B as input, and generates a specialized program that computes P's result given C, D. Suppose we are given Gen_P, but not P. The goal is to find a new generating extension Gen_R of a program R such that for all A, B, C, D:

$$\langle\langle Gen_R\ A, C\rangle\ B, D\rangle = \langle\langle Gen_P\ A, B\rangle\ C, D\rangle. \qquad (2)$$

where $\langle R\ A, C, B, D\rangle = \langle P\ A, B, C, D\rangle$. We call such a problem *the regeneration problem of* Gen_P. Note that B, C are consumed at different stages in Eq. (2). It is clear that the regeneration problem stated here can be generalized to multi-level generating extensions [5]. In this paper however, we focus on the two-level case.

It is well known that the *Futamura projections* (FMP) [2] can turn a general program, say P, into a generating extension, e.g. Gen_P, by program specialization (Eq. (1)). The question has been raised whether, in principle, one can modify the staging of a generating extension, say Gen_P, in a similar fashion (Eq. (2)). However, none of the existing projections covers the latter case: neither the FMP, nor the *specializer projections* (SPP) [3], nor the *degeneration projections* (DGP) [7] can be used. The projections can be applied sequentially to achieve the effect of regeneration in particular cases, but practical experience suggests that the generation of intermediate programs is not (always) a good idea [5]. Hence, we are looking for a *direct* and *general* solution to the regeneration problem.

Generally, it seems hard to reason about multiple levels of program transformers and to describe applications that go beyond the self-application of the classical FMP, such as the regeneration problem considered here. This paper adopts the language-independent notions of [4,6], based upon Turchin's metasystem transition [10], to achieve full formalization and clarity with minimal means. It helped us quite a bit in understanding and describing symbolically the different kinds of objects and operations performed on them.

This paper takes the idea of program transformation by metasystem transition further. A *regeneration scheme* is formulated which shows how to modify the staging of a generating extension by a two-level metasystem structure using *program specialization* and *program composition*. Distinguishing between these two equivalence transformations allows us to identify certain abstract properties needed to solve the problem under consideration.

From the results described in this paper, we can see that *program generation* and *program degeneration*, which superficially seem very different, are two extremes of the *same* transformation dimension. Indeed the standard projections for program generation and degeneration, FMP, SPP and DGP, can all be shown to be instances of the regeneration scheme. Finally, it appears that an understanding of some of the deeper issues underlying metasystem structures in program transformation is beginning to emerge.

We assume familiarity with the basic notions of partial evaluation, in particular the Futamura Projections and self-application (a good source is [8]).

2. Preliminaries

We now formulate the properties of generating extensions, program specializers and program composers more precisely; see [4] for details.

Data, programs and application. We assume a fixed set of data for input and output, and for representing programs written in different languages. To express syntactically the *application* of an L-program to its input we use angular brackets $\langle\ldots\rangle_{\mathsf{L}}$ (we omit the language index L when it is not essential). Computation (reduction) of an application expression is denoted by $\Rightarrow$. For example, $\langle P\ X, Y\rangle_{\mathsf{L}} \Rightarrow Out$ is the reduction of program P with input X, Y to output Out. 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 C-program Int is an A/C-*interpreter* if for every A-program P and every input X: $\langle Int\ P, X\rangle_{\mathsf{C}} = \langle P\ X\rangle_{\mathsf{A}}$.

Definition 2 (*translator*). A C-program $Trans$ is an A→B/C-*translator* if for every A-program P and every input X: $\langle\langle Trans\ P\rangle_{\mathsf{C}}\ X\rangle_{\mathsf{B}} = \langle P\ X\rangle_{\mathsf{A}}$.

Definition 3 (*generating extension*). A C-program Gen_P is a B/C-*generating extension* of an A-program P if for every input X, Y: $\langle\langle Gen_P\ X\rangle_{\mathsf{C}}\ Y\rangle_{\mathsf{B}} = \langle P\ X, Y\rangle_{\mathsf{A}}$.

3. Equivalence transformation as metacomputation

Abstraction. An application expression including variables, called *configuration* [9], 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, $\langle P\ \ X, y\rangle$ is a configuration where y is a variable.

Encoding. To represent configurations as data, we assume an injective mapping, called *metacoding*, from configurations into 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. $\overline{\langle P\ \ X, y\rangle}$ is a metacoded configuration.

Metaprograms. It follows from our notation that $\langle Meta\ \overline{\langle P\ \ X, y\rangle}\rangle \Rightarrow Out$ denotes *metacomputation* on a metacoded configuration by metaprogram *Meta*. We refer to any process of simulating, analyzing or transforming programs by means of programs as metacomputation. For better readability, we move metacoded expression one line down for each metacoding (called *MST-schemes*; first suggested by Turchin):

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

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 the 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 $\overline{X}$). Repeated use of abstraction, encoding, and metacomputation leads to a hierarchy of metaprograms. For example,

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

We are now in the position to define an important class of metaprograms, namely equivalence transformers, and to specify two transformation tasks. The metanotation $C\sigma$ denotes the result of applying substitution σ to configuration C.

Definition 4 (*equivalence transformer*). A program *Meta* is an *equivalence transformer* if for all substitutions $\sigma = \{x_1 \mapsto X_1, \ldots, x_n \mapsto X_n\}$ where $x_1, \ldots, x_n$ are all variables that occur free in a configuration C:[3]

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

Definition 5 (*program specializer*). An equivalence transformer *Spec* written in C is an A→B/C-*specializer* if for every A-program P, every input X, Y:

$$\langle\langle Spec \underset{\langle P\ \ X, y\rangle_{\mathsf{A}}}{\underline{\qquad\quad}}\rangle_{\mathsf{C}}\ Y\rangle_{\mathsf{B}} = \langle P\ \ X, Y\rangle_{\mathsf{A}}.$$

Definition 6 (*program composer*). An equivalence transformer *Comp* written in C is an A→B/C-*composer* if for every A-program P, Q, every input X, Y:

$$\langle\langle Comp \underset{\langle P\ \langle Q\ \ x\rangle_{\mathsf{A}}, y\rangle_{\mathsf{A}}}{\underline{\qquad\qquad\quad}}\rangle_{\mathsf{C}}\ X, Y\rangle_{\mathsf{B}} = \langle P\ \langle Q\ \ X\rangle_{\mathsf{A}}, Y\rangle_{\mathsf{A}}.$$

The two definitions reveal the close connection between program specialization and program composition: composition of a program P with a constant program Q, that is $\langle Q\rangle \Rightarrow X$, amounts to specialization. On the other hand, we do not expect a specializer to perform (non-trivial) composition of programs.

4. The regeneration scheme

The regeneration scheme which we will define now tells us how to change the computation staging of a generating extension by specialization and composition. It shows that a generating extension expecting one part of the input early, can be regenerated to produce a new generating extension expecting another

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

$$\boxed{\begin{array}{l}\hfill\text{MST-formula}\hfill\\ \hline \langle \mathit{Spec}\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\rangle_{\mathsf{E}} \Rightarrow \mathit{Gen}'_R \\ \quad\langle \mathit{Comp}\underline{\qquad\qquad a \qquad c \qquad}\rangle_{\mathsf{D}} \\ \qquad\qquad\langle \mathit{Int}\ \langle \mathit{Gen}_P\ \bullet, b\rangle_{\mathsf{C}}, \bullet, d\rangle_{\mathsf{C}}\end{array}}$$

Fig. 1. The regeneration scheme for Gen_P.

part of the input early. The general case of multi-language regeneration is considered.

(i) Suppose we are given a $\mathsf{B/C}$-generating extension Gen_P of an A-program P and a $\mathsf{B/C}$-interpreter *Int*. Hence, for all input A, B, C, D:

$$\langle Int\ \langle Gen_P\ A, B\rangle_{\mathsf{C}}, C, D\rangle_{\mathsf{C}} = \langle P\ A, B, C, D\rangle_{\mathsf{A}}. \tag{3}$$

(ii) Assume further two equivalence transformers, a $\mathsf{C}{\to}\mathsf{F/D}$-composer *Comp* and a $\mathsf{D}{\to}\mathsf{G/E}$-specializer *Spec*.

The objective is to find an (efficient) $\mathsf{F/G}$-generating extension Gen'_R of a program R such that for all input A, B, C, D:

$$\langle\langle Gen'_R\ A, C\rangle_{\mathsf{G}}\ B, D\rangle_{\mathsf{F}} = \langle\langle Gen_P\ A, B\rangle_{\mathsf{C}}\ C, D\rangle_{\mathsf{B}}. \tag{4}$$

We present the regeneration scheme for finding Gen'_R.

(1) Given A, C define a specialized program P_{AC} such that

$$\langle P_{AC}\ B, D\rangle_{\mathsf{C}} = \langle\langle Gen_P\ A, B\rangle_{\mathsf{C}}\ C, D\rangle_{\mathsf{B}} \tag{5}$$

for all B, D. For every Gen_P, A, C there exists a (trivial) C-program P_{AC} as shown constructively by the definition

$$\textbf{def}\ \langle P_{AC}\ b, d\rangle_{\mathsf{C}} \triangleq \langle Int\ \langle Gen_P\ A, b\rangle_{\mathsf{C}}, C, d\rangle_{\mathsf{C}},$$

where P_{AC}'s computation is performed in two stages: first by producing a program using Gen_P, then interpreting the new program with *Int*. In other words, *Int* provides an interpretation of Gen_P's output. The two stages together recover P's original functionality. A (more efficient) program P'_{AC} can be obtained by applying the composer *Comp* to P_{AC}'s definition in order to remove the computational overhead caused by the generation and subsequent interpretation of an intermediate program:

$$\langle Comp\underset{\langle Int\ \langle Gen_P\ A, b\rangle_{\mathsf{C}}, C, d\rangle_{\mathsf{C}}}{\underline{\qquad\qquad\qquad\qquad\qquad}}\rangle_{\mathsf{D}} \Rightarrow P'_{AC}$$

From Definition 6 we have equation

$$\langle P'_{AC}\ B, D\rangle_{\mathsf{F}} = \langle P_{AC}\ B, D\rangle_{\mathsf{C}}. \tag{6}$$

(2) Define a generating extension Gen_R such that

$$\langle\langle Gen_R\ A, C\rangle_{\mathsf{D}}\ B, D\rangle_{\mathsf{F}} = \langle P'_{AC}\ B, D\rangle_{\mathsf{F}}. \tag{7}$$

for all A, B, C, D. For every Gen_P there exists a (trivial) D-program Gen_R as shown constructively by the definition

$$\textbf{def}\ \langle Gen_R\ a, c\rangle_{\mathsf{D}} \triangleq \langle Comp\underset{\langle Int\ \langle Gen_P\ \bullet, b\rangle_{\mathsf{C}}, \bullet, d\rangle_{\mathsf{C}}}{\underline{\qquad\qquad a \qquad c \qquad}}\rangle_{\mathsf{D}}.$$

A (more efficient) generating extension Gen'_R can be obtained by applying the specializer *Spec* to Gen_R's definition in order to specialize the composer *Comp* with respect to *Int* and Gen_P. The new scheme, which we call *the regeneration scheme*, is shown in Fig. 1. From Definition 5 we have

$$\langle Gen'_R\ A, C\rangle_{\mathsf{G}} = \langle Gen_R\ A, C\rangle_{\mathsf{D}}. \tag{8}$$

Theorem 7. *Program Gen'_R obtained by the regeneration scheme* (*Fig.* 1) *using the programs specified in* (i) *and* (ii) *is an* $\mathsf{F/G}$-*generating extension such that Eq.* (4) *holds for all* A, B, C, D.

Proof. The correctness of the regeneration scheme follows from its construction above. Eq. (9) follows from Eqs. (5) and (6). From Eqs. (7) and (8) we have Eq. (10). Combining both equations it follows that Gen'_R is the $\mathsf{F/G}$-generating extension we are looking for in Eq. (4).

$$\langle P'_{AC}\ B, D\rangle_{\mathsf{F}} = \langle\langle Gen_P\ A, B\rangle_{\mathsf{C}}\ C, D\rangle_{\mathsf{B}}, \tag{9}$$

$$\langle\langle Gen'_R\ A, C\rangle_{\mathsf{G}}\ B, D\rangle_{\mathsf{F}} = \langle P'_{AC}\ B, D\rangle_{\mathsf{F}}. \qquad \square \tag{10}$$

Note that the regeneration scheme does not use self-application as does the 2nd FMP, but a more general

	Conversion	MST-formula
a	*Trans* $\rightarrow$ *Mix*	$\begin{array}{l}\langle Spec \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \rangle_{\mathsf{E}} \Rightarrow Mix \\ \quad \langle Comp \_\_\_\_\_\_\_\_ p \_\_\_ x \_\_\_ \rangle_{\mathsf{D}} \\ \qquad \langle Sint\ \langle Trans\ \bullet\rangle_{\mathsf{C}}, \bullet, y\rangle_{\mathsf{C}}\end{array}$
b	*Mix* $\rightarrow$ *Trans*	$\begin{array}{l}\langle Spec \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \rangle_{\mathsf{E}} \Rightarrow Trans \\ \quad \langle Comp \_\_\_\_\_\_\_\_ p \_\_\_\_\_\_\_\_ \rangle_{\mathsf{D}} \\ \qquad \langle Sint\ \langle Mix\ \bullet, x\rangle_{\mathsf{C}}, y\rangle_{\mathsf{C}}\end{array}$

Fig. 2. Direct regeneration: translator to specializer and vice versa.

	Type of equation	Characterization	Example
a	$\langle\langle Gen_P\ A\rangle\ C\rangle = \langle\langle Gen_R\ A, C\rangle\rangle$	Full generation	-
b	$\langle\langle Gen_P\ A\rangle\ C, D\rangle = \langle\langle Gen_R\ A, C\rangle\ D\rangle$	Partial generation	FMP, SPP
c	$\langle\langle Gen_P\ A\rangle\ D\rangle = \langle\langle Gen_R\ A\rangle\ D\rangle$	Identity regeneration	-
d	$\langle\langle Gen_P\ A, B\rangle\ D\rangle = \langle\langle Gen_R\ A\rangle\ B, D\rangle$	Partial degeneration	-
e	$\langle\langle Gen_P\ B\rangle\ D\rangle = \langle\langle Gen_R\rangle\ B, D\rangle$	Full degeneration	DGP

Fig. 3. Characterization of Gen_P's five regeneration problems.

metasystem structure of equivalence transformers and abstractions. The difference is that regeneration uses a composer to transform the composition of *Int* and Gen_P, while the 2nd FMP requires "only" the transformation power of a specializer, and uses a simpler abstraction pattern (it does not involve a generating extension Gen_P; see Proposition 10).

The program Gen'_R is an F/G-generating extension which inherits its target language F from the composer and its implementation language G from the specializer. The languages B, C, D, and E are arbitrary in the regeneration scheme: they disappear during the transformation process.

5. A regeneration problem

It is known that translators and specializers are generating extensions of interpreters [2,3]. We now consider a regeneration problem, namely the conversion of a translator into a specializer, and vice versa.

Suppose we have two C-programs, an A→C/C-translator *Trans* and a C/C-interpreter *Sint*. Hence, for every A-program P and every input X, Y:

$$\langle Sint\ \langle Trans\ P\rangle_{\mathsf{C}}, X, Y\rangle_{\mathsf{C}} = \langle P\ X, Y\rangle_{\mathsf{A}}. \qquad (11)$$

The objective is to find an A→C/C-specializer *Mix* such that

$$\langle\langle Mix\ P, X\rangle_{\mathsf{C}}\ Y\rangle_{\mathsf{C}} = \langle\langle Trans\ P\rangle_{\mathsf{C}}\ X, Y\rangle_{\mathsf{C}}. \qquad (12)$$

It is easy to see that the generating extensions *Mix* and *Trans* in Eq. (12) correspond to the generating extensions Gen_R and Gen_P in Eq. (4), respectively. Thus, the problem under consideration can be solved by the regeneration scheme: specializing the composition of the translator and the interpreter. Since we have B = C = F = G for the languages in Eq. (4), the following two transformers are needed: a C→C/D-composer *Comp* and a D→C/E-specializer *Spec*. Languages D, E are arbitrary and, thus, two source-to-source transformers may be used instead by letting C = D = E. Fig. 2(a) shows the conversion of *Trans* into *Mix* using the regeneration scheme where $a \equiv p, c \equiv x, d \equiv y$ (and omitting b). Similarly, Fig. 2(b) shows the conversion of *Mix* into *Trans*.

6. Characterization of regeneration problems

It is characteristic for regeneration that Gen_R's staging is different from Gen_P's staging. For clarity we used four arguments in the regeneration scheme,

Step	Conversion	MST-formula	Note
1	$Mix \rightarrow Int'$	$\begin{array}{c}\langle Comp \underline{\qquad\qquad\qquad}\rangle_{\mathsf{C}} \Rightarrow Int' \\ \langle Sint\ \langle Mix\ p, x\rangle_{\mathsf{C}}, y\rangle_{\mathsf{C}}\end{array}$	1st DGP
2	$Int' \rightarrow Trans$	$\begin{array}{c}\langle Spec \underline{\qquad\qquad\qquad}\rangle_{\mathsf{C}} \Rightarrow Trans \\ \langle Spec \underline{\quad} p \underline{\quad}\rangle_{\mathsf{C}} \\ \langle Int'\ \bullet, x, y\rangle_{\mathsf{C}}\end{array}$	2nd FMP

Fig. 4. Indirect regeneration: specializer to translator.

but it is easy to see that each of them, say A, can be replaced by a (possibly empty) sequence, e.g. $A_1, \ldots, A_n$ ($n \geqslant 0$). Consequently, Gen_R may have more or less arguments than Gen_P, i.e. some input will be consumed "earlier" or "later", respectively. This leads us to a characterization of five regeneration problems (Fig. 3). Moreover, we shall now see that the standard projections, FMP, SPP, and DGP, are instances of the regeneration scheme.

We say that P is a *constant program* if it has no input, that is $\langle P\rangle = C$. A program Gen_P is a *constant generating extension* of a program P if Gen_P is a constant program and for all X: $\langle\langle Gen_P\rangle\ X\rangle = \langle P\ X\rangle$.

(a) *Full generation*: Gen_R is a generating extension which, given all input, returns a constant program.

(b) *Partial generation*: Gen_R has more arguments than Gen_P, but does not return a constant program. In other words, some input is consumed earlier. The conversion of a translator into a specializer is such a case (cf. Eq. (12)). Moreover, the FMP and SPP are special cases of partial generation as shown below.

(c) *Identity regeneration*: Gen_R has the same input as Gen_P. This case covers, among others, the transformation of an A→B/C-translator into an A→F/C-translator by letting C = G in Eq. (4) ("retargeting"), and the transformation of a B/C-generating extension into a B/C-generating extension by letting B = F, C = G. Depending on the programs used for regeneration, the latter transformation may modify the intensional behavior of a generating extension while leaving its extensional behavior unchanged.

(d) *Partial degeneration*: Gen_R has fewer arguments than Gen_P, but Gen_R is not a constant generating extension. In other words, some of Gen_P's input is consumed later. The conversion of a specializer into a translator is such a case (cf. Eq. (12)).

(e) *Full degeneration*: Gen_R has no arguments, i.e. Gen_R is a constant generating extension, $\langle Gen_R\rangle = R$. The conversion of a translator or a specializer into an interpreter is such a case (cf. Example 9). As we shall see below, the DGP are a special case of full degeneration.

Proposition 8. *The regeneration scheme (Fig. 1) reduces to the first DGP [7] if Gen'_R is a constant generating extension of an* F*-program R.*

Proof. To ensure that Gen'_R is a constant generating extension, we omit variables a, c from the regeneration scheme. In the resulting scheme, $Spec$ operates on a ground configuration (no free variables) and can be removed since by Def. 4 we have $\langle\langle Spec\ \overline{\mathcal{G}}\rangle\rangle = \mathcal{G}$ for all ground configurations $\mathcal{G}$. The resulting scheme is the 1st DGP:

$$\begin{array}{c}\langle Comp \underline{\qquad\qquad\qquad}\rangle_{\mathsf{D}} \Rightarrow R. \qquad \square \\ \langle Int\ \langle Gen_P\ b\rangle_{\mathsf{C}}, d\rangle_{\mathsf{C}}\end{array}$$

Example 9. Consider the conversion of an A→C/C-specializer Mix (Eq. (12)) into an A/C-interpreter Int' such that $\langle Int'\ P, X, Y\rangle_{\mathsf{C}} = \langle\langle Mix\ P, X\rangle_{\mathsf{C}}\ Y\rangle_{\mathsf{C}}$. Assume a C→C/C-composer $Comp$ and a C/C-interpreter $Sint$. Full degeneration of Mix into Int' according to the 1st DGP is shown in Fig. 4(1).

Proposition 10. *The regeneration scheme (Fig. 1) reduces to the second FMP [2] if Gen_P is a constant generating extension of a* C*-program P.*

Proof. To ensure that Gen_P is a constant generating extension, we omit variables a, b from the regeneration scheme. Since Gen_P is a constant generating extension of a C-program P we have $\langle P\ C, D\rangle_{\mathsf{C}} = \langle Int\ \langle Gen_P\rangle_{\mathsf{C}}, C, D\rangle_{\mathsf{C}}$ for all C, D. Using this equality from right to left, we resolve the composition problem at the lowest level.[4] Thus, *Comp* can be replaced by a (simpler) equivalence transformer: a specializer *Spec'*. The resulting scheme is the 2nd FMP (self-application of *Spec* requires $\mathsf{C} = \mathsf{D} = \mathsf{E}$, $\mathsf{F} = \mathsf{G}$):

$$\langle Spec \underset{\langle Spec' \underset{\langle P \ \bullet, d\rangle_{\mathsf{C}}}{\underline{\quad c \quad}} \rangle_{\mathsf{D}}}{\underline{\qquad\qquad\qquad}} \rangle_{\mathsf{E}} \Rightarrow Gen'_R. \qquad \square$$

Example 11. Consider the conversion of an A/C-interpreter *Int'* (Example 9) into an A→C/C-translator *Trans*. Assume further a (self-applicable) source-to-source transformer, namely a C→C/C-specializer *Spec*. Partial generation of *Int'* into *Trans* according to the 2nd FMP is shown in Fig. 4(2). Schemes (1) and (2) applied sequentially achieve the effect of partial degeneration, namely the conversion of *Mix* into *Trans* (Section 5), but in contrast to the regeneration scheme this transformation requires two separate steps.

Proposition 12. *The regeneration scheme (Fig. (1)) reduces to the second SPP [3] if Gen_P is a constant generating extension of a* C*-program P and c is split into two parts.*

Proof. Follows the proof of Proposition 10, resulting in the 2nd SPP:

$$\langle Spec \underset{\langle Spec' \underset{\langle P \ \bullet, \bullet, d\rangle_{\mathsf{C}}}{\underline{\quad c_1\text{-}c_2 \quad}} \rangle_{\mathsf{D}}}{\underline{\qquad\qquad\qquad}} \rangle_{\mathsf{E}} \Rightarrow Gen'_R. \qquad \square$$

7. A regenerator as generating extension

We complete the presentation, noting that the regeneration scheme suggests the existence of a *regenerator*, a tool for changing the computation staging of a generating extension.

Definition 13 (*regenerator*). A program *Regen* is a *regenerator of generating extensions* if for every generating extension Gen_P of P, every input A, B, C, D:

$$\langle\langle\langle Regen\ Gen_P\rangle\ A, C\rangle\ B, D\rangle = \langle\langle Gen_P\ A, B\rangle\ C, D\rangle.$$

[4] In fact, it is already a specialization problem since Gen_P is a constant program.

How can one obtain a regenerator? We know of two construction methods using equivalence transformers (beside hand-writing a regenerator). First, similar to the 3rd FMP, a regenerator can be obtained by abstracting from program Gen_P in the regeneration scheme (Fig. 1) and applying a specializer to the new configuration. Second, we can view a regenerator as a generating extension of a program *Dint*, called *degenerating interpreter* [7], satisfying $\langle Dint\ Gen_P, A, B, C, D\rangle = \langle P\ A, B, C, D\rangle$ for all generating extensions Gen_P of P. It follows that

$$\langle\langle\langle Regen\ Gen_P\rangle\ A, C\rangle\ B, D\rangle$$
$$= \langle Dint\ Gen_P, A, B, C, D\rangle.$$

This suggests that a regenerator *Regen* is a three-level generating extension of a degenerating interpreter *Dint* and, thus, *Regen* can be obtained from *Dint* using (multi-level) specialization [5]. This is remarkable because one does not need a "universal" composer, but can resolve the composition problem already in the (manual) construction of *Dint*. It may be easier to write *Dint* than to design a general composer that specializes well with respect to (arbitrary) interpreters. In other words, specialization is sufficient to obtain *Regen* from *Dint*.

8. Discussion

When a generating extension is constructed, it is vital to know which part of the input will be supplied early, and which late. We have shown that the computation staging of a generating extension can be changed by the regeneration scheme using three programs: an interpreter, a specializer, and a composer.

Several points about the regeneration scheme must be made. The scheme says nothing about the quality of the generated programs. It exhibits the underlying metasystem structure of the problem and tells us what can be achieved given (strong enough) specializers and composers. Both are equivalence transformers and thus, from an extensional view, are exchangeable. However, they are software tools and may be targeted towards different applications. For instance, it has been shown that partial evaluation techniques, based on aggressive constant propagation, are powerful enough to cover a large field of important applications [8], while they are not good enough to achieve non-trivial program composition as in [9,11]. Thus it is not only conceptually productive to distinguish between these transformation tasks. It allows to identify certain abstract properties needed to solve a specific transformation problem (e.g. the layers of composition and specialization). Finally, note that the full computational realization of the regeneration scheme is still an open problem – although partial solutions already exist, namely for the FMP and SPP [8,5].

References

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

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

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

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

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

[6] R. Glück and A.V. Klimov, Metasystem transition schemes in computer science and mathematics, *World Futures* 45 (1995) 213–243.

[7] R. Glück and A.V. Klimov, On the degeneration of program generators by program composition, *New Generation Computing*, to appear.

[8] N.D. Jones, C.K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation* (Prentice-Hall, New York, 1993).

[9] V.F. Turchin, The concept of a supercompiler, *Trans. Programming Languages Systems* 8 (3) (1986) 292–325.

[10] V.F. Turchin, Metacomputation: Metasystem transitions plus supercompilation, in: O. Danvy, et al., eds., *Partial Evaluation*, Lecture Notes in Computer Science, Vol. 1110 (Springer, Berlin, 1996) 481–509.

[11] P. Wadler, Deforestation: Transforming programs to eliminate trees, *Theoret. Comput. Sci.* 73 (1990) 231–248.