J. Functional Programming 4 (4): 499–514, October 1994 © 1994 Cambridge University Press

On the generation of specializers[1]

ROBERT GLÜCK

Institut für Computersprachen, University of Technology Vienna, A-1040 Vienna, Austria

Abstract

Self-applicable specializers have been used successfully to automate the generation of compilers. Specializers are often rather sophisticated, for which reason one would like to adapt and transform them with the aid of the computer. But how to automate this process? The answer to this question is given by three *specializer projections*. While the Futamura projections define the generation of compilers from interpreters, the specializer projections define the generation of specializers from interpreters. We discuss the potential applications of the specializer projections, and argue that their realization is a real touchstone for the effectiveness of the specialization principle. In particular, we discuss generic specializers, bootstrapping of subject languages and the generation of optimizing specializers from interpretive specifications. The Futamura projections are regarded as a special case of the specializer projections. Recent results confirm that the specializer projections can be performed in practice using partial evaluators.

Capsule review

Program specialization or partial evaluation has by now more than proven its worth as a realistic program transformation paradigm. The *Futamura projections* stand as the cornerstone of the development of program specialization. These projections tell us that if we write a *self-applicable* program specializer we get much more than just a specializer: we get the possibility to generate compilers from interpreters, and even a standalone compiler generator.

This paper takes the idea of the Futamura projections even further. It defines three variations of the Futamura projections called the *specializer projections*.

Where the first Futamura projection tell us how to generate target programs from an interpreter and a subject program, the first specializer projection tells us how, in the additional presence of some of the input to the subject program, to generate a specialized version of the subject program. Likewise, where the second Futamura projection tell us how to generate compilers, the second specializer projection tells us how to generate specializers. Finally, where the third Futamura projection tells us how to generate compiler generators, the third specializer projection tells us how to generate specializer generators.

So if we are given self-applicable specializer and we write an interpreter we get much more than just an interpreter: we get the possibility to specialize programs in another language, and to generate both a new specializer for this language, and a specializer generator.

[1] Supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (No. J0780). Current address: DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark (*e-mail*: glueck@diku.dk).

1 Introduction

The driving force behind the investigation of program specialization and self-application are the three *Futamura projections* (Futamura, 1971). These projections assert that compilers and compiler generators can be obtained from interpreters by *self-application* of a program specializer; an approach which has many practical advantages (Jones, 1990; Pagan, 1988). Partial evaluation, a technique for program specialization, has been very successful due to its promising applications and recent advances both in theory and practice. Fully automatic, self-applicable partial evaluators now exist for several languages, and have been used for generating compilers according to the Futamura projections (e.g. Andersen 1992; Bondorf and Danvy, 1991; Consel and Danvy, 1991; Jones *et al.*, 1989; Jørgensen, 1992; Launchbury, 1991; Romanenko, 1988).

Current research in program specialization aims at extending the state of the art in two directions: improving the specialization of programs and adapting existing methods to new language features and paradigms. Improving program specializers is not a trivial task: it requires considerable insights into an existing system or building a new system from scratch. Specializers are often rather sophisticated, for which reason we would like to adapt and transform them automatically. While self-applicable specializers have been used to generate compilers and other program generators, the generation of specializers by self-application seems far away. Indeed, the Futamura projections give us no clue on how to generate specializers from interpreters.

In this paper, we assert that specializers can be generated by self-application. As in the Futamura projections, we define three *specializer projections*. While the Futamura projections allow the generation of compilers from interpreters, the specializer projections enable the generation of specializers from interpreters. The specializer projections apply the methods of program specialization to the problem of specializer construction. Constructing specializers is just another instance of the constant struggle between universality and specialization. Recent results confirm that the specializer projections can be performed in practice, even with existing partial evaluators (Glück and Jørgensen, 1994).

Given a self-applicable $R \to R$-specializer and the interpretive definition of a language S in R one can perform $S \to R$-specialization, generate an $S \to R$-specializer and an $S \to R$-specializer generator. As a consequence, one does not need to devise $S_i \to R$-specializers for different subject language S_i $(i = 1, \ldots, n)$ individually, just to provide the interpretive definition of R. Other possible applications of the specializer projections include (i) using *generic specializers*; (ii) *bootstrapping* of subject languages, and (iii) generating *optimizing specializers* from appropriate interpreters (i.e. to generate specializers that are stronger than the self-applicable specializer used in their generation).

This paper is organized as follows. After reviewing the background and notation in section 2, the specializer projections are defined in section 3. In section 4 we consider the most general case of multilanguage specializer generation, and discuss the inheritance of languages and functionality in the generation process. In section 5

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

we outline three applications of the specializer projections. Section 6 discusses practical aspects, and section 7 concludes with related work.

2 Background

Program specialization is a program transformation principle for specializing programs with incomplete input information. Given a *subject program* S and some partially known input information, a program specializer α generates a *residual program* R that returns the same result when given the remaining input information as S when applied to the complete input. The ultimate goal of program specialization is to improve the efficiency of the subject program by exploiting the known information in advance. The resulting residual program R can be much faster than the subject program S. The reader is referred to the literature for more technical details (Bjørner *et al.*, 1988; Jones *et al.*, 1993).

Let S be an S-program of two arguments, and let D_1, D_2 be two input values. Running S-program S with the input D_1, D_2 is written as follows (S represents the complete program text; the subscript **S** indicates the language in which the program is written)

$$\texttt{Result} = S_S(D_1, D_2)$$

The result of specializing the S-program S with respect to D_1 by an S→R-specializer is an R-program R that returns the same result (if it exists) when given the remaining input D_2 as the S-program S

$$S_S(D_1, D_2) = R_R(D_2)$$

A program specializer α can be described as a program with three parameters (Jones *et al.*, 1989): a subject program, a string classifying the arguments of the subject program as either known (static, 's') or unknown (dynamic, 'd'), and a list of the known input values. The classification is used to make it explicit which arguments of a subject program are known/unknown.

Definition An *I*-program α is an *S→R-specializer* iff for all S-programs S and input values D_1, D_2

$$S_S(D_1, D_2) = R_R(D_2) \quad \textit{where} \quad R_R = \alpha_I(S_S, \texttt{'sd'}, [D_1])$$

In case the subject program S is equal to α, we speak of *self-application*. It can easily be seen that for self-application to be feasible, *I* must be equal to S (or, at least, a subset of S). Most self-applicable program specializers are S→S-specializers written in S, where S = subset of a high-level language (such as Lisp, Prolog or C). Few S→R-specializers have been implemented (e.g. AMIX, where S = subset of Lisp; R = code for a stack-machine (Holst, 1988)). We depict specializers using T-diagrams where the arrow is marked with a Greek letter (to distinguish specializers from compilers). Figure 1 illustrates an S→R-specializer written in *I*.

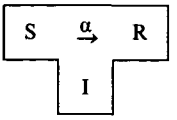

Fig. 1. T-diagram of an S→R-specializer written in I

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

2.1 Futamura projections

The *Futamura projections* (Futamura, 1971) assert that compilers and compiler generators can be obtained from interpreters by self-application of a program specializer. Several self-applicable program specializers have been built and applied according to the Futamura projections (Jones *et al.*, 1993). The formulas are termed 'projections' according to Ershov (1982), who coined the term for denoting the specialization of programs with respect to some fixed argument (in an analytical geometry sense). The following programs will be used in the Futamura projections (and later in the specializer projections):

- A self-applicable R → R-specializer α written in R.
- An S-interpreter `Int` written in R.

For notational convenience we assume that all interpreted S-programs have two arguments (of course, each argument may be a list of several arguments). This assumption makes it easier to compare the Futamura projections and the specializer projections. We define `Int` to be an interpreter with three parameters: the interpreted program `S`, and the two arguments of `S`.

Definition An R-program `Int` is an S-Interpreter iff for all S-programs `S` and input values D_1, D_2

$$S_S(D_1, D_2) = Int_R(S_S, D_1, D_2)$$

The *1st Futamura projection* follows directly from the definition of the R → R-specializer α and the S-Interpreter `Int`. The effect of S → R-compilation is achieved by specializing the S-interpreter `Int` with respect to an S-program `S`. During the specialization of the S-interpreter the S-program `S` is static while both of its arguments are dynamic (therefore the classification of the S-interpreter `Int` is `'sdd'`). The result is the residual program `T`.

$$T_R = \alpha_R(Int_R, \text{'sdd'}, [S_S]) \qquad S \to R\text{-compilation}$$

According to the definition of the R → R-specializer α and the S-interpreter `Int`, the new R-program `T` returns the same result as the S-program `S` with the same input: $S_S(D_1, D_2) = Int_R(S_S, D_1, D_2) = R_R(D_1, D_2)$, i.e. the R-program `T` is a compiled version of the S-program `S`.

The *2nd Futamura projection* follows from the 1st projection by applying the specializer α a second time (the first self-application takes place), i.e. by specializing the specializer α with respect to the S-interpreter `Int` and its classification `'sdd'`. The result is the S → R-compiler `C`. According to the definition of α, the same target program `T` can be obtained by applying the new program `C` to the S-program `S` (instead of using the 1st Futamura projection), i.e. the program `C` resulting from the 2nd Futamura projection corresponds to an S → R-compiler

$$C_R = \alpha_R(\alpha_R, \text{'ssd'}, [Int_R, \text{'sdd'}]) \qquad S \to R\text{-compiler generation}$$

The *3rd Futamura projection* follows from the 2nd projection by applying the specializer α a third time (the second and final self-application takes place). The

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

specializer α is now specialized with respect to α. The result is the compiler-generator G. According to the definition of the S→R-specializer α, the S→R-compiler C can be generated by applying the program G to the S-Interpreter Int and the classification 'sdd' (instead of using the 2nd Futamura projection), i.e. the program G resulting from the 3rd Futamura projection corresponds to a compiler generator

$G_R = \alpha_R(\alpha_R, \text{'ssd'}, [\alpha_R, \text{'sdd'}])$ *S→R-compiler-generator generation*

3 Specializer projections

The specializer projections assert that, given a self-applicable R→R-specializer α and the interpretive definition of language S in R, one may perform S→R-specialization, generate an S→R-specializer and an S→R-specializer generator. The following programs will be used in the three specializer projections (the same programs as in the Futamura projections, see above):

- A self-applicable R→R-specializer α written in R.
- An S-interpreter Int written in R.

3.1 1st specializer projection

The 1st specializer projection follows directly from the definition of the R→R-specializer α and the S-interpreter Int. The effect of S→R-specialization is achieved by specializing the S-interpreter Int with respect to an S-program S and – in contrast to the 1st Futamura projection – some part of S's input. Assume that the input D_1 is given at specialization time (i.e. the classification of the S-interpreter is 'ssd'). The residual program R is defined by

$$R_R = \alpha_R(Int_R, \text{'sdd'}, [S_S, D_1])$$

According to the definition of the R→R-specializer α and the S-interpreter Int the new R-program R returns the same result given the remaining input D_2 as the S-program S with the complete input D_1, D_2. Obviously, the R-program R is a specialized version of the S-program S

$$S_S(D_1, D_2) = Int_R(S_S, D_1, D_2) = R_R(D_2)$$

i.e. the effect of S→R-specialization can be achieved without an S→R-specializer. An efficient residual program R may be produced if all operations in the interpreter Int depending only on the program S and the static input D_1 can be executed at specialization time.

3.2 2nd specializer projection

The 2nd specializer projection follows from the 1st projection by applying the specializer α a second time, i.e. by specializing the specializer α with respect to the interpreter Int and the classification 'ssd'

$$\gamma_R = \alpha_R(\alpha_R, \text{'ssd'}, [Int_R, \text{'ssd'}])$$

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

According to the definition of α, the R-program R can be obtained by applying the new R-program γ ('generated') to the S-program S and the input D_1

$$R_R = \gamma_R(S_R, D_1).$$

The program γ corresponds to an *S→ R specializer*: applying γ to an S-program S and a part of its input D_1 produces an R-program R which is a specialized version of S. That is, the 2nd specializer projection asserts the generation of an S→ R-specializer by specializing an R→ R-specializer with respect to an S-interpreter.

3.3 3rd specializer projection

The 3rd specializer projection follows from the 2nd projection by applying the specializer α a third time, i.e. the specializer α is specialized with respect to α. The result is an R-program G which is a specialized version of α

$$G_R = \alpha_R(\alpha_R, \text{'sdd'}, [\alpha_R, \text{'ssd'}])$$

According to the definition of the S→ R-specializer α the S→ R-specializer γ can be generated by applying the program G to the S-interpreter Int and the classification 'ssd'

$$\gamma = G_R(Int_R, \text{'ssd'})$$

The residual program G can be called an *S→ R-specializer generator*, i.e. the 3rd specializer projection defines the generation of an S→ R-specializer generator by double self-application of an R→ R-specializer α.

3.4 Specializer projections and Futamura projections

The specializer projections can be summarized as follows:

$R_R = \alpha_R(Int_R, \text{'ssd'}, [S_S, D_1])$ — *S→ R-specialization*

$\gamma_R = \alpha_R(\alpha_R, \text{'ssd'}, [Int_R, \text{'ssd'}])$ — *S→ R-specializer generation*

$G_R = \alpha_R(\alpha_R, \text{'ssd'}, [\alpha_R, \text{'ssd'}])$ — *S→ R-specializer-generator generation*

Applying the generated programs to the corresponding input

$$\begin{aligned} S_S(D_1, D_2) &= R_R(D_2) \\ &= \gamma_R(S_R, D_1)\ (D_2) \\ &= G_R(Int_R, \text{'ssd'})\ (S_S, D_1)\ (D_2) \end{aligned}$$

The building material used in the specializer projections is the same as in the Futamura projections. The difference between the S→ R-compilation asserted by the 1st Futamura projection and the S→ R-specialization asserted by the 1st specializer projection is that in the latter case, the S-interpreter is also specialized with respect to some part of the input to the S-program. The difference between compiler generation and specializer generation in the 2nd projections is the classification of the S-interpreter during specialization time. The 3rd projections coincide. In fact, the generator G is a general currying function which can be used with varying

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

classifications. For example, the generator `G` can be used for generating an S→R-compiler from the S-interpreter `Int` given the classification `'sdd'` and for generating an S→R-specializer given the classification `'ssd'`.

The Futamura projections may be regarded as a special case of the specializer projections: with D_1 being 'empty' the specializer projections are reduced to the Futamura projections. This is not surprising, since S→R-compilation (no input is given in advance) can be viewed as a special case of S→R-specialization (some input is given in advance). If we have solved the problem of specializer generation, we have also solved the problem of compiler generation.

There are now two ways for using self-application: specializer generation or compiler generation (as illustrated in Fig. 2). In contrast to the Futamura projections,

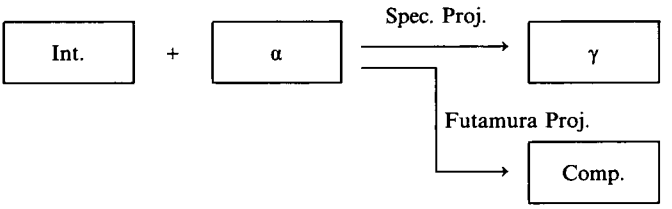

Fig. 2. Two ways using self-application: specializer generation and compiler generation

the specializer projections can be applied 'repeatedly': the new specializer γ can be reused according to the specializer projections (or the Futamura projections). This is not possible with a compiler generated by the Futamura projections.

An open question is how far the generation of S→R-specializers can be taken. How 'close' must the languages S and R be that the generation of S→R-specializers is practical? Could S be a 'high-level' language while R is a 'low-level' language? These questions also remain to be answered for the Futamura projections.

3.5 Self-application of a generated S→R-specializer

The specializer projections define how to generate an S→R-specializer from an S-interpreter and an S→R-specializer. It may be desirable to self-apply the new specializer according to the Futamura projections (or the specializer projections). In section 2 we argued that self-application requires the specializer to be written in its own subject language. However, the specializer γ cannot be self-applied because it is written in R while its subject language is S (the incompatibility of γ's self-application is illustrated in Fig. 3*a*). How to use the new specializer γ? The solution is simple:

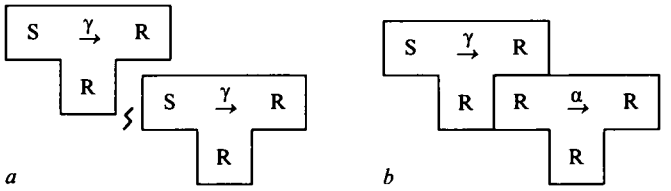

Fig. 3 (*a*). Incompatible self-application of γ; (*b*) quasi self-application

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

reuse the initial R→R-specializer α in the projections instead of attempting the self-application of γ (Fig. 3*b*). In other words, the new S→R-specializer γ is not self-applied (and does not need to be self-applicable). Such a combination of specializers is usually not used for generating program generators because it requires to implement two specializers. However, in our case one specializer is generated from another specializer.

4 On the generation of specializers

Two important aspects of specializers generation: (1) the languages (subject, residual, implementation language), and (2) the functionality which the new specializers inherit. In this section we assume that two specializers and an interpreter are given: an A→Y-specializer α, a B→Z-specializer β and an X-interpreter `Int`. According to the specializer projections, a new specializer γ may be generated

$$\gamma_Z = \beta_C(\alpha_B, \text{'ssd'}, [\text{Int}_A, \text{'ssd'}])$$

4.1 The languages of the generated specializers

The new specializer γ is an X→Y-specializer written in Z. It inherits the three languages X, Y and Z from the initial building material (as can easily be verified):

- Its subject language X from the interpreter `Int`.
- Its residual language Y from the specialized specializer α.
- Its implementation language Z from the specializing specializer β.

The three other languages A, B and C – implementation language of the interpreter and the subject languages of the two specializers – can be chosen arbitrarily: they disappear during the generation process. (cf. Fig. 4). Note that the specializers α and β are not self-applied.

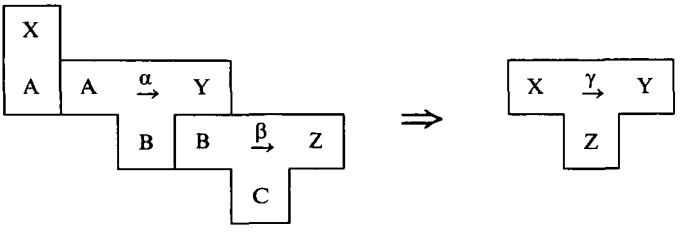

Fig. 4. Multilanguage specializer generation

The inheritance of the languages can be represented in the following scheme of *multilanguage specialization*. The T-diagrams are simply abbreviated (e.g. the specializer α is characterized as A→Y/B). Note that this scheme is also valid for generating compilers according to the Futamura projections. Since we are interested in the language inheritance scheme we omit other details

$$\boxed{X/A + A \to Y/B + B \to Z/C = X \to Y/Z}$$

There are three interesting instances of the above scheme:

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

- S→R-specializer generation by self-application of an R→R-specializer (cf. section 3.2)

$$S/R + R\to R/R + R\to R/R = S\to R/R$$

- Specializer generation using two specializers, an S→R- and an R→R-specializer (cf. section 3.5)

$$L/S + S\to R/R + R\to R/R = L\to R/R$$

- Self-generation of an R→R-specializer using a self-interpreter (cf. section 5.3)

$$R/R + R\to R/R + R\to R/R = R/R$$

4.2 The functionality of the generated specializers

The new specializer γ inherits not only the three languages X, Y and Z but also its functionality from the initial building material. The two specializers and the interpreter play different roles in the generation of γ.

- the specializing specializer β implements the 'machinery' of γ;
- γ inherits its functionality from the specialized specializer α and the interpreter `Int`.

We should stress that the specializer projections assert nothing about the efficiency or specialization power of the specializers. The quality and efficiency of the generated specializer γ depends on the transformations and strategies built into the original specializers α and β, and on the interpreter `Int` (cf. section 5.3). One would expect (and hope) that those criteria for the effectiveness of specializers, such as Jones' (1990) optimality criterion, which have been satisfied 'pragmatically' for hand-written specializers would also hold for generated specializers. However, such properties cannot be guaranteed in general, since it is impossible to construct (and to generate) specializers that guarantee the maximum use of information for non-trivial programming languages. For the specializer projections to be of practical value, the techniques involved and the results must be 'strong enough'. Ideally one would hope to achieve (one of) the following goals:

(a) specializer γ be at least as efficient as the original specializers;
(b) specializer γ be at least as strong as the original specializers.

It is interesting to note that recent investigations demonstrated the feasibility of the second goal (Glück and Jørgensen, 1994; Turchin, 1993).

5 Approaches to specializer development

There are several techniques that enable us to make good use of the specializer projections. In this section, we suggest three methods which aim at the modification of the subject languages and the extension of the functionality of specializers.

5.1 Generic specializers

The specializer projections enable $L_i\to R$-specializers to be generated for different languages $L_i (i = 1, \ldots, n)$, given a self-applicable R→R-specializer α. The economy of

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

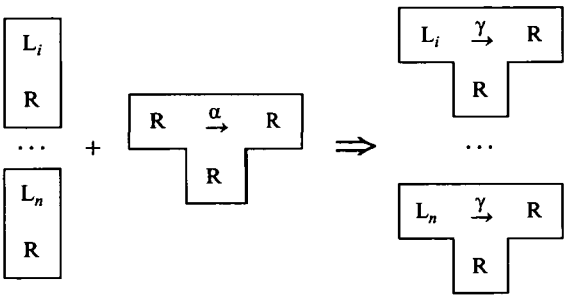

Fig. 5. Using a generic specializer α

this approach is obvious: a *generic specializer* can be devised and reused for different languages. One does not need to write $L_i \rightarrow$ R-specializers for each new subject language, just give their interpretive definition in R. It is much easier to construct a generic specializer plus a set of L_i-interpreters than to write all $L_i \rightarrow$ R-specializers by hand. This method allows us to separate the definition of a language from the specialization proper. A similar approach was used to derive manually a new specialization algorithm for Prolog by writing its evaluation function in a functional language and using an existing algorithm for minimal function graphs that was already given for the functional language (Gallagher and Bruynooghe, 1991).

The 'parallel' application of the specializer projections (Fig. 5) is the simplest technique using the specializer projections for automating the construction (and maintenance) of $L_i \rightarrow$ R-specializers. In case the generic specializer is modified, all specializers can be automatically generated again. Only the generic specialization engine needs to be updated and maintained. In addition, the same set of L_i-interpreters can be used together with different generic specializers. Note that the $L_i \rightarrow$ R-specializers generated from the same specializer have the same residual language (this also holds for the $L_i \rightarrow$ R-compilers generated from the same specializer using the Futamura projections).

5.2 Bootstrapping

The specializer projections may be used to *bootstrap* the subject language S of an S → R-specializer by defining extended language features in a subset of the language by an interpretive definition. This is related to the well-known technique of bootstrapping compilers, but different in the use of interpretive definitions.

First, an R → R-specializer is constructed, where R could be either an initial subset S_0 of the subject language S or some language in which we would like to write an S_0-interpreter in. Then the specializer for the full subject language S can be generated step-by-step by supplying a series of interpretive definitions S_1 in S_0, S_2 in S_1 ... (where S_i is a subset of S_{i+1}) and by repeatedly applying the specializer projections until the full language S is reached. Each step in the process of bootstrapping can profit from the previous steps. New features are expressed in terms of simpler ones. This is the essence of the bootstrapping process; using the facilities offered by a subset of S to definite more advanced features of S.

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

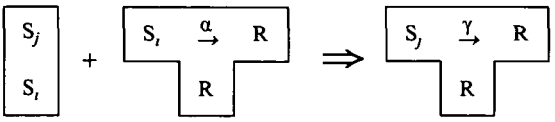

Fig. 6. A step during the bootstrapping of a subject language

The bootstrapping technique allows the incremental development of specializers. The advantage is obvious: it is usually much easier to construct a specializer for a subset of a language S than to build a specializer for full S. The implementation of the extensions is left to the computer.

Such a 'serial' application of the specializer projections (Fig. 6) allows to develop specializers in a modular and systematic way, i.e. by devising independent interpretive definitions for each extension of the subject language. Each definition can be incorporated automatically into a specializer. This simplifies the construction and the maintenance of specializers: simple add or modify the corresponding interpretive definitions. In case the underlying specialization engine is modified, the specializers can be generated again from the existing definitions. This makes it possible to separate specific language features from the specialization kernel.

5.3 'Bricks instead of tricks'

Generated specializers inherit not only their languages but also their functionality from the initial specializers and interpreters. In the preceding sections, the specializer projections were used to generate specializer for various subject languages. The same principles can be applied to the construction of *optimizing specializers*, which enables us to generate more efficient residual programs. So far, two approaches are known that enable one to influence the way in which subject programs are specialized:

(1) rewrite the subject program (cf. Consel and Danvy, 1989);
(2) rewrite the specializer (cf. Bondorf, 1992).

The specializer projections offer a new alternative:

(3) provide an interpretive definition of the subject language.

Recent investigations confirm the feasibility of the third approach (cf. section 6.2): it was shown that interpreters may be used for improving the transformational power of supercompilation (Turchin, 1993), and that generated specializers may be stronger than the partial evaluator used for their generation (Glück and Jørgensen, 1994). Using the *interpretive approach*, one may achieve a better specialization of programs without rewriting each program individually, as in case (1), and without understanding how an existing specializer works internally, as in case (2). Instead of manually extending a specializer with numerous 'tricks', one could implement corresponding methods in interpreters and generate the corresponding optimizing specializers, i.e. each 'trick' may be put into a 'brick') One can also imagine a

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

collection of such interpreters being shared, reused and combined to derive various sophisticated specializers.

It is obvious that the three approaches discussed in this section can be combined in various ways. For example, one could imagine a 'specializer tool kit' including a library of generic specializers, a collection of 'tricks in bricks' and interpreters for various subject languages.

6 Two practical approaches

In this section, we will outline how one can achieve in practice what has been discussed in principle. Assume that the same 'building material' as in the definition of the specializer projections is given (section 3): an S-interpreter `Int` and a self-applicable R → R-specializer α, both written in R.

6.1 Incremental S → R-specialization

The effect of S → R-specialization can be achieved by splitting the task into two steps:

(1) S → R-translation of an S-program `S` into an R-program `R`.
(2) R → R-specialization of the R-program `R` with respect to known input D_1.

Obviously, the specializer used in the first step has to produce target programs written in the subject language of the specializer used in the second step; this is the case for the R → R-specializer α (of course, one could use two different specializers provided the first specializer produces programs written in the subject language of the second specializer). A program `Spec` can be defined for performing S → R-specialization in two steps. Assume that `Spec` is written in the language R. The program may look like

```
define Spec (int,p,d1)
    let q := α(int,'sdd',[p])          ;S→R-translation (1st step)
     in α(q,'sd',[d1])                 ;R→R-specialization (2nd step)
```

A call to `Spec` correspond to the 1st specializer projection: given an S-interpreter, and S-program and some of its input `Spec` returns to specialized version of the S-program written in R. To improve the specialization process defined by `Spec` one can specialize the program `Spec` with respect to a particular interpreter `Int`. This corresponds to the 2nd specializer projection. Applying the specializer α to `Spec` converts the first stage into an S → R-compiler (according to the 2nd Futamura projection). However, not much will be improved by specializing the second stage, since neither the input `q` nor `d1` are known for α. The 3rd specializer projection over `Spec` basically inserts a compiler generator in the first stage. Not much can achieved in the second stage.

A possible optimization is the generation of an optimizing compiler for the first stage provided an appropriate interpreter is given (Jørgensen, 1992). One can expect that the specialization of optimized target programs will then return more efficient residual programs in the second stage. However, a drawback of this method is that the optimizing compiler will not be able to use the information provided in the second

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

stage, i.e. ultimately the specialization power of `Spec` is determined by the strength of the specializer used in the second stage.

Note that in case the specializer α is a partial evaluator directed by a binding-time analysis, such as Mix (Jones *et al.*, 1989), the specialization process defined above incorporates two binding-time analysis: (i) the binding-time analysis of the interpreter `Int` in the first stage (for guiding the compilation of the subject program); (ii) the binding-time analysis of the target program produced by the first stage in the second stage (for guiding the specialization of the target program).

Splitting the specialization process is related to the two-step method used for semantics-directed compiler generation (Jones, 1988). In fact, both methods use *incremental specialization* (Glück, 1991b).

6.2 Direct $S \rightarrow R$-specialization

The 'natural' approach to specializer generation is to follow the specializer projections directly, i.e. specializing the S-interpreter `Int` with respect to an S-program and know parts of its input. Practical results using this approach and an existing partial evaluator (i.e. Similix) are reported elsewhere (Glück and Jørgensen, 1993). The interpreter `Int` was written carefully to ensure that the input to the interpreter S-program can be separated into two parts (corresponding to the static and dynamic input). Moreover, it was confirmed that the generated specializers may be stronger than the partial evaluator α used for their generation (cf. section 5.3), i.e. the combined effect of the partial evaluator α and an interpreter `Int` may be greater than the effect of the specializer α alone. To the best of our knowledge, these are the first practical results using the specializer projections for generating optimizing specializers.

7 Related work

The specializer projections and their potential applications have been anticipated for some time (Glück, 1991*a*). They are an instance of the more general principle of metasystem transition. Other potential applications of metasystem transition have been discussed elsewhere (Abramov, 1991; Glück, 1991*b*, 1992; Turchin, 1980, 1986). Using multiple interpretive layers for program transformation was suggested by Turchin (1993) and called the *stairway effect*: a 'mechanical' way to repeat metasystem transitions.

It is well-known that optimizing compilers can be generated from interpreters by self-application of a program specializer. An optimizing compiler producing very efficient program code for a substantial subset of a realistic lazy functional language was obtained by providing an appropriate interpreter (Jørgensen, 1991, 1992). The properties of the partial evaluator and the way in which the interpreter was written ensured that all compiled programs had certain properties, e.g. efficient pattern matching and no code duplication. From this viewpoint, it is not surprising that interpreter can be also be used for optimizing the specialization of programs.

The residual language of the generated specializer is fixed. Changing the residual language requires retargeting a specializer manually (automatic retargeting would

https://doi.org/10.1017/S0956796800001167 Published online by Cambridge University Press

require the use of program inversion (Abramov, 1991; Romanenko, 1991), which is beyond the scope of this paper).

8 Conclusion

Whereas the Futamura projections allow us to generate compilers from interpreters, the specializer projections enable us to generate specializers from interpreters. Self-applicable specializers have been used successfully for generating compilers. However, specializers are often rather sophisticated, for which reason one would also like to adapt and transform them automatically. The specializer projections apply the methods of program specialization to the problem of specializer construction. How can we convince others if we do not use our own methods to solve our own problems (i.e. the construction of specializers)?

We have presented three approaches using the specializer projections for the development of specializers: generic specializers, bootstrapping of subject languages, and using interpreters to optimize specialization. Common to all three approaches is the use of interpretive specifications. This has several advantages compared to the manual construction of specializers: it is generally easier to read and write interpreters; the optimizations in the interpreters are integrated into the specializer automatically; debugging an interpreter is simpler than debugging a specializer; and one can instrument an interpreter with operations tracing the execution or doing statistics (which results in a specializer doing these operations). In conclusion, the specializer projections offer potentially more flexibility and economy in the development of specializers.

Recent results confirm that specializers can indeed be generated from interpreters, even with existing self-applicable partial evaluators. This can be seen as a first step towards automatizing the construction of specializers. These results are also a strong evidence that one should not limit the specialization principle to the generation of compilers. The specializer projections can be seen as another 'self-application': the use of specializers to generate specializers.

Using the specializer projections has, beyond practical advantages, the following interesting property: the correctness of the generated specializer is guaranteed by the correctness of the generic specializer and the correctness of the interpreter.

Specializer generation is new, and it is impossible to say how far this approach can be taken. Some intriguing questions remain: How much can be achieved by a repeated application of the specializer projections? What is the minimal functionality an initial specializer must provide? How far these principles can be extended and what their limitations are, will be a challenging problem for future research.

Acknowledgements

This work could not have been carried out without the pioneering work of Valentin Turchin on metasystem transition. It is a pleasure to acknowledge the presentation and discussion of this material at the 'NYU Partial Computation and Program Analysis Day' in June 1991. I greatly appreciate stimulating discussions with the members of the Refal group in Moscow in September 1991. I would especially like

to thank Andrei Klimov, Arkady Klimov, Alexander Romanenko and Sergei Romanenko. Special thanks are due to Sergei Romanenko for thorough comments on an earlier paper, and for his encouragement.

It is a pleasure to acknowledge useful discussions with Anders Bondorf, Carsten K. Holst, Neil Jones and Jesper Jørgensen. I would also like to thank the DIKU TOPPS group for providing an excellent working environment.

References

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

Andersen, L. O. (1992) Self-applicable C program specialization. In: *Proceedings Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. San Francisco, CA, pp. 54–61.

Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.) (1988) *Partial Evaluation and Mixed Computation*. North-Holland.

Bondorf, A. (1992) Improving binding times without explicit CPS-conversion. In: *ACM Conference on Lisp and Functional Programming*. San Francisco, CA, 1–10.

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

Consel, C. and Danvy, O. (1989) Partial evaluation of pattern matching in strings. *Information Processing Letters* **30** (2) 79–86.

Consel, C. and Danvy, O. (1991) Static and dynamic semantics processing. In: *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*. Orlando, FL, 14–24. ACM Press.

Ershov, A. P. (1982) Mixed computation: potential applications and problems for study. *Theoretical Computer Science* **18** 41–67.

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

Gallagher, J. and Bruynooghe, M. (1991) The derivation of an algorithm for program specialization. *New Generation Computing* **9** (3–4) 305–333.

Glück, R. (1991*a*) *On the generation of* $S \rightarrow R$ *specializers*. Technical Report, University of Technology, Vienna. (Presented at the NYU Partial Computation and Program Analysis Day. June 21 1991, New York.)

Glück, R. (1991*b*) Towards multiple self-application. In: *Proceedings Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. New Haven, CT, 309–320. ACM Press.

Glück, R. (1992) Projections for knowledge based systems. In: Trappl R (ed.), *Cybernetics and Systems Research '92. Vol.* 1, pp. 535–542, World Scientific.

Glück, R. and Jørgensen, J. (1994) Generating optimizing specializers. In: IEEE International Conference on Computer Languages, Toulouse, France, IEEE Computer Society Press (to appear).

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

Jones, N. D. (1988) Challenging problems in partial evaluation and mixed computation. In: Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.), *Partial Evaluation and Mixed Computation*. Gammel Avernæs, Denmark. 1–14, North-Holland.

Jones, N. D. (1990) Partial evaluation, self-application and types. In: Paterson M. S. (ed.), *Automata, Languages and Programming. Warwick University, UK, Lecture Notes in Computer Science, Vol. 443*, pp. 639–659, Springer-Verlag.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

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

Jørgensen, J. (1991) Generating a pattern matching compiler by partial evaluation. In: Peyton Jones, S. L., Hutton, G. and Holst, C. K. (eds), *Proceedings Glasgow Workshop on Functional Programming*, Ullapool, Scotland, pp. 177–195, Springer–Verlag.

Jørgensen, J. (1992) Generating a compiler for a lazy language by partial evaluation. In: *Conference Record of the Nineteenth Symposium on Principles of Programming Languages.* Albuquerque, NM, pp. 258–268. ACM Press.

Launchbury, J. (1991) A strongly-typed self-applicable partial evaluator. In: Hughes J. (ed.), *Functional Programming Languages and Computer Architecture.* Cambridge, MA, pp. 145–164, Springer-Verlag.

Pagan, F. G. (1988) Converting interpreters into compilers. *Software – Practice and Experience* **18** 509–527.

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

Romanenko, S. (1988) A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In: Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.), *Partial Evaluation and Mixed Computation.* Gammel Avernæs, Denmark, pp. 445–463, North-Holland.

Turchin, V. F. (1980) The use of metasystem transition in theorem proving and program optimization. In: de Bakker, J. W. and van Leeuwen J. (eds.), *Automata, Languages and Programming: Noordwijkerhout, Netherlands, Lecture Notes in Computer Science, Vol. 85,* pp. 645–657, Springer-Verlag.

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

Turchin, V. F. (1993) Program transformation with metasystem transitions. *Journal of Functional Programming* **3** (3) 283–314, July.