The Translation Power of the Futamura Projections

Robert Glück *

PRESTO, JST & Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan, glueck@acm.org

Abstract. Despite practical successes with the Futamura projections, it has been an open question whether target programs produced by specializing interpreters can always be as efficient as those produced by a translator. We show that, given a Jones-optimal program specializer with static expression reduction, there exists for every translator an interpreter which, when specialized, can produce target programs that are at least as fast as those produced by the translator. This is not the case if the specializer is not Jones-optimal. We also examine Ershov's generating extensions, give a parameterized notion of Jones optimality, and show that there is a class of specializers that can always produce residual programs that match the size and time complexity of programs generated by an arbitrary generating extension. This is the class of generation universal specializers. We study these questions on an abstract level, independently of any particular specialization method.

1 Introduction

Program specialization, also known as partial evaluation, has shown its worth as a realistic program transformation paradigm. The *Futamura projections* [5] stand as the cornerstone of the development of program specialization. They capture a fundamental insight, namely that programs can be translated by specializing an interpreter. Several program specializers which perform these transformations automatically have been designed and implemented [14].

Despite these practical successes, it has been an open question (*e.g.*, [5, 12, 16]) whether target programs produced by specializing interpreters can be as efficient as those produced by traditional translators, or whether there exist certain theoretical limitations to Futamura's method. We study these questions on an abstract level, independently of any particular specialization method or other algorithmic detail. We are interested in statements that are valid for all specializers and for all programming languages.

The results in this paper expand on previous work on Jones optimality [3, 8, 13, 17, 23, 24]. First, we show that, given a Jones-optimal program specializer with static expression reduction, there exists for every translator an interpreter

* On leave from DIKU, Dept. of Computer Science, University of Copenhagen.

M. Broy and A.V. Zamulin (Eds.): Perspectives of System Informatics. Proceedings. LNCS 2890, pp. 133–147, 2003.

which, when specialized, can produce target programs that are at least as fast as those produced by the translator. This is the class of *translation universal* specializers. We also show that a specializer that is not Jones-optimal is not translation universal. Finally, we introduce a parameterized notion of Jones optimality and show that there is a class of *generation universal* specializers that can always produce residual programs that match the size and time complexity of programs generated by an arbitrary generating extension. Previously, it was found intuitively [12, 13] that Jones optimality would be a "good property". The results in this paper give formal status to the term "optimal" in the name of that criterion.

Our results do not imply that a specializer that is not Jones-optimal is not useful, but that there exists a *theoretical limit* to what can be achieved in general. They do not tell us when program generation by a specializer is preferable over program generation by a generating extension (parser generator, translator, *etc.*). This depends on pragmatic considerations and on the application at hand. We hope that the investigations in this paper will lead to a better understanding of some of the fundamental issues involved in automatic program specialization.

This paper is organized as follows. After reviewing standard definitions of several metaprograms (Sect. 2), we discuss translation by specialization (Sect. 3), Jones optimality (Sect. 4), and the degeneration of generating extensions (Sect. 5). Then, we present our main results on translation (Sect. 6) and program generation (Sect. 7). We discuss related work (Sect. 8) and finally conclude (Sect. 9). We assume that the reader is familiar with the ideas of partial evaluation, *e.g.*, as presented in [14, Part II].

2 Fundamental Concepts

What follows is a review of standard definitions of interpreters, translators, and specializers. We also formalize a notion of static expression reduction and define certain measures on programs. The notation is adapted from [14], except that we use divisions (SD) when classifying the parameters of a program as static or dynamic. We assume that we are dealing with universal programming languages.

We briefly review some basic notation. For any program text, p, written in language L, we let $[\![p]\!]_L\, d$ denote the application of L-program p to its input d. Multiple arguments are written as a list, such as $[\![p]\!]_L\, [\, d_1, ..., d_n\,]$. The notation is strict in its arguments. Programs and their input and output are drawn from the same data domain D. This is convenient when dealing with metaprograms which take programs and data as input. We define a program domain $P_L \subseteq D$, but leave the details of language L unspecified. Let $P_L^p \subseteq P_L$ denote the set of all L-programs that are functionally equivalent to L-program p. Equality ($=$) shall always mean strong (computational) equivalence: either both sides are defined and equal, or both sides are undefined. When we define a program using λ-notation, the text of the corresponding L-program is denoted by $\ulcorner \lambda ... \urcorner_L \in P_L$. This mapping is fixed. The intended semantics of the λ-notation is call-by-value.

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

$$[\![int]\!]_L\,[\, p, d\,] \;=\; [\![p]\!]_N\; d\ .$$

Definition 2 (self-interpreter). *An L-program* $sint \in Sint_L$ *is a* self-interpreter for L *iff sint is an* L/L*-interpreter.*

Definition 3 (translator). *An L-program* $trans \in Trans_{N\to L}$ *is an* $N \to L$-translator *iff* $\forall p \in P_N, \forall d \in D$*:*

$$[\![\, [\![trans]\!]_L\, p]\!]_L\, d \;=\; [\![p]\!]_N\; d\ .$$

Definition 4 (generating extension). *An L-program* $gen \in Gen_L^p$ *is an* L-generating extension of a program $p \in P_L$ *iff* $\forall x, y \in D$*:*

$$[\![\, [\![gen]\!]_L\, x]\!]_L\, y \;=\; [\![p]\!]_L\,[\, x, y\,]\ .$$

Definition 5 (specializer). *An L-program* $spec \in Spec_L$ *is an* L-specializer *iff* $\forall p \in P_L, \forall x, y \in D$*:*

$$[\![\, [\![spec]\!]_L\,[\, p, \mathrm{SD}, x\,]]\!]_L\, y \;=\; [\![p]\!]_L\,[\, x, y\,]\ .$$

For simplicity, we assume that the programs that we specialize have two arguments $[\, x, y\,]$, and that the first argument is static. Even though Def. 5 makes use of only one division 'SD', a constant, we keep the division argument. The program produced by specializing p is a *residual program*. A specializer does not need to be total. Def. 5 allows *spec* to diverge on input $[\, p, \mathrm{SD}, x\,]$ iff p diverges on input $[\, x, y\,]$ for all y. Actual specializers often sacrifice the termination behavior implied by this definition and terminate less often.

A specializer is *trivial* if the residual programs produced by the specializer are simple instantiations of the source programs. This trivial specializer never improves the efficiency because the source programs remain unchanged.

Definition 6 (trivial specializer). *An L-specializer* $spec_{triv} \in Spec_L$ *is* trivial *iff* $\forall p \in P_L, \forall x \in D$*:*

$$[\![spec_{triv}]\!]_L\,[\, p, \mathrm{SD}, x\,] \;=\; \ulcorner \lambda y.[\![p]\!]_L\,[\, x, y\,] \urcorner_L\ .$$

More realistic specializers should evaluate as many static expressions as possible to improve the efficiency of the residual programs. However, an investigation [11] suggests that for universal languages, one cannot devise a specializer that makes '*maximal use*' of the static input in all programs. This makes it difficult to give a general definition of static expression reduction. Here, we define a special case that can be decided, namely static arguments that are always evaluated.

Definition 7 (static expression reduction). *An L-specializer* $spec \in Spec_L$ *has* static expression reduction *iff* $\forall p, q, q' \in P_L$*,* $\forall x \in D$*:*

$$\begin{gathered} p \;=\; \ulcorner \lambda(a, b).[\![q]\!]_L\,[\, [\![q']\!]_L\, a, b\,] \urcorner_L \;\Longrightarrow \\ [\![spec]\!]_L\,[\, p, \mathrm{SD}, x\,] \;=\; [\![spec]\!]_L\,[\, q, \mathrm{SD}, [\![q']\!]_L\, x\,] \end{gathered}$$

This definition tells us that there is no difference between specializing a program q with respect to a static value $[\![q']\!]\,x$ and specializing program p, the composition of programs q and q', with respect to a static value x. We expect that both residual programs are identical if *spec* fully evaluates the static expression $[\![q']\!]\,a$ in p.[1] Most specializers that have been implemented reduce more static expressions in p than what is required by Def. 7, for example, supported by a binding-time analysis as in offline partial evaluation [14].

A specializer with static expression reduction is not trivial; it can be viewed as a *universal program* (it is capable of simulating an arbitrary program q'). It cannot be total. A total specializer must be conservative enough to terminate on every static input.

Finally, we introduce two measures on programs, namely *running time* and *program size*, and define two partial orders on programs, which we will use to discuss the transformation power of a specializer. The measure of the running time of a program can be based on a timed semantics (*e.g.*, counting the number of elementary steps a computation takes) [14]. The measure of program size can be the size of one of its representations (*e.g.*, the abstract syntax tree).

Definition 8 (time order). *For program $p \in P_L$ and data $d \in D$, let $Time(p, d)$ denote the* running time *to compute $[\![p]\!]_L\, d$. The partial order $\leq_{Time}$ on L-programs is defined by*

$$p \leq_{Time} q \stackrel{\text{def}}{=} \forall d \in D\colon Time(p, d) \leq Time(q, d)\ .$$

Definition 9 (size order). *For program $p \in P_L$, let $Size(p)$ denote the* size *of p. The partial order $\leq_{Size}$ on L-programs is defined by*

$$p \leq_{Size} q \stackrel{\text{def}}{=} Size(p) \leq Size(q)\ .$$

The corresponding equivalence relations, $=_{Time}$ and $=_{Size}$, are defined in the obvious way (by $=$ instead of $\leq$).

3 Translation by Specialization

The *Futamura projections* [5] stand as the cornerstone of the development of program specialization. They tell us how to translate programs by specializing interpreters, how to generate stand-alone translators and even a generator of translators. Here we discuss the 1st Futamura projection. A detailed account of all three projections can be found in [14].

Let p be an N-program, let *int* be an N/L-interpreter, and let *spec* be an L-specializer. Then the *1st Futamura projection* is defined by

$$p' \;=\; [\![spec]\!]_L\,[\,int, \mathrm{SD}, p\,]\ . \tag{1}$$

[1] Programs q and q' can also be viewed as a decomposition of a program p into static and dynamic components. A method that transforms a program into a functionally equivalent program with strongly separated binding times can be found in [19].

Here, the interpreter *int* is specialized with respect to program p. Using Defs. 1 and 5, we have the functional equivalence between p' and p:

$$[\![p']\!]_L\, d \;=\; [\![int]\!]_L\,[\,p, d\,] \;=\; [\![p]\!]_N\, d\ . \tag{2}$$

We observe that p is written in language N, while p' is written in language L. This means that an N-to-L-translation was achieved by specializing *int* with respect to p. We say that p' is a *target program* of *source program* p.

A trivial translation is always possible using a trivial specializer. The first argument of *int* is instantiated to p, and the result is a trivial target program:

$$[\![spec_{triv}]\!]_L\,[\,int, \mathrm{SD}, p\,] \;=\; \ulcorner \lambda d.[\![int]\!]_L\,[\,p, d\,]\urcorner_L\ . \tag{3}$$

The target program on the right hand side of (3) is inefficient: it contains an interpreter. Running this target program will be considerably slower than running the source program p, often by an order of magnitude. This is not the translation we expect.

Thus, a natural goal is to produce target programs that are at least as fast as their source programs. To achieve this goal, it is important that the specializer removes the *entire* interpretation overhead. Unfortunately, we cannot expect a specializer to achieve this optimization for *any* interpreter. As mentioned in Sect. 2, there exists no specializer that could make 'maximal use' of the static input in all programs, here, in all possible N/L-interpreters.

4 Jones Optimality

A specializer is said to be "strong enough" [13] if it can completely remove the interpretation overhead of a self-interpreter. The following definition is adapted from [14, Sect. 6.4]. It makes use of the 1st Futamura projection discussed in the previous section, but uses a self-interpreter instead of an N/L-interpreter.

Definition 10 (Jopt-$\leq_{Time}$). *Let* $sint \in Sint_L$ *be a self-interpreter, then a specializer* $spec \in Spec_L$ *is* Jones-optimal for *sint* with respect to $\leq_{Time}$ *iff* $Jopt_{\leq_{Time}}(spec, sint)$ *where*

$$Jopt_{\leq_{Time}}(spec, sint) \stackrel{\text{def}}{=} \forall p \in P_L : [\![spec]\!]_L\,[\,sint, \mathrm{SD}, p\,] \leq_{Time} p\ .$$

A specializer *spec* is said to be *Jones-optimal* if there exists a self-interpreter *sint* such that, for all source programs p, the target program produced by specializing the self-interpreter is at least as fast as p. This tells us that *spec* can remove the interpretive overhead. The case of *self-interpreter specialization* is interesting because it is easy to judge to what extent the interpretive overhead has been removed by comparing the source and target programs, as they are written in the same language. In particular, when both programs are identical up to variable renaming, it is safe to conclude that this goal is achieved. When it is clear from the context, we will refer to the definition above simply as Jones optimality.

Remark. Jones optimality of a specializer is easier to state than to achieve. It was shown [20] that a specializer for a first-order functional language with algebraic data types requires a combination of several non-trivial transformation methods to become Jones-optimal (*e.g.*, partially static structures, type specialization). The first implementation of a Jones-optimal specializer was an offline partial evaluator for a Lisp-like language [21]. Jones optimality was first proven [23] for lambda-mix. Recent work [3, 24] has focused on the problem of tag-elimination.

Existence of Jones-optimal Specializers. As discussed in [12], a Jones-optimal specializer can be built by a simple construction that just returns its third argument in case the first argument is textually identical to a particular self-interpreter (here, *mysint*); otherwise it performs a trivial specialization. Such a specializer can be defined by

$$\begin{aligned} spec_{jopt} \stackrel{\text{def}}{=} \ulcorner\lambda(p, \mathrm{SD}, x)\,.\, &\mathit{if}\ \mathit{equal}(p, \mathit{mysint})\ \mathit{then}\ x \\ &\mathit{else}\ [\![spec_{triv}]\!]_L\,[\, p, \mathrm{SD}, x\,]\urcorner_L \end{aligned} \tag{4}$$

This construction is not useful in practice, but it is sufficient to show that a Jones-optimal specializer can be built for every programming language. The specializer satisfies $Jopt_{\leq_{Time}}(spec_{jopt},\ mysint)$.

Next, we show that there exists a Jones-optimal specializer with static expression reduction. This fact plays an important role later in this paper. Such a specializer can be defined by

$$\begin{aligned} spec_{jser} \stackrel{\text{def}}{=} \ulcorner\lambda(p, \mathrm{SD}, x)\,.\, &\mathit{case}\ p\ \mathit{of} \\ \text{`}\lambda(a, b).[\![q]\!]\,[\,[\![q']\!]\ a, b\,]\text{'} &\rightarrow [\![spec_{jser}]\!]_L\,[\, q, \mathrm{SD}, [\![sint]\!]_L\,[\, q', x\,]\,] \\ \mathit{otherwise} &\rightarrow [\![spec_{jopt}]\!]_L\,[\, p, \mathrm{SD}, x\,]\urcorner_L \end{aligned} \tag{5}$$

where *case* implements pattern matching: if program p consists of a composition[2] of two programs q and q', then q is specialized with respect to the result of evaluating $[\![sint]\!]\,[\, q', x\,]$ where *sint* is a self-interpreter; otherwise p is specialized with respect to x. Assume that program *mysint* does not suit the decomposition in *case*. Then we have $Jopt_{\leq_{Time}}(spec_{jser},\ mysint)$ since for all L-programs p:

$$[\![spec_{jser}]\!]_L\,[\, mysint, \mathrm{SD}, p\,] \;=\; p\ . \tag{6}$$

The specializer has the static expression reduction property of Def. 7 since for all L-programs $p = \ulcorner\lambda(a, b).[\![q]\!]_L\,[\,[\![q']\!]_L\ a, b\,]\urcorner_L$ and for all data x:

$$[\![spec_{jser}]\!]_L\,[\, p, \mathrm{SD}, x\,] \;=\; [\![spec_{jser}]\!]_L\,[\, q, \mathrm{SD}, [\![q']\!]_L\ x\,]\ . \tag{7}$$

Remark. More realistic Jones-optimal specializers, for instance [18, 20, 21, 23], do not rely on any of the constructions above. For several of these specializers, the target programs are *identical, up to variable renaming*, to the source programs. These specializers, as well as $spec_{jopt}$ and $spec_{jser}$ shown above, satisfy $Jopt_{=_{Time}}$,

[2] We shall not be concerned with the technical details of parsing L-programs.

which we obtain by replacing $\leq_{Time}$ by $=_{Time}$ in Def. 10. There exists no specializer that would satisfy $Jopt_{<_{Time}}$. This would imply that we can improve the running time of a program infinitely often by repeated specialization, which is not possible – just consider an empty program that contains no operations.

5 Degeneration of Generating Extensions

The notion of a *generating extension* [4] describes uniformly the functioning of various program generators. This is a powerful concept because it captures the essence of different program generators, including diverse applications such as parsing, graphics, and pattern matching. Turning a general program into a generating extension has been intensively studied in the area of partial evaluation [14].

Consider the converse goal: turning a generating extension back into a general program. Given an L-generating extension *gen*, we can always construct a program p', a *degeneration of gen*, such that

$$[\![p']\!]_L\,[\,x, y\,] \;=\; [\![\,[\![gen]\!]_L\,x\,]\!]_L\,y\ . \tag{8}$$

Let *sint* be a self-interpreter for L. Then we can immediately define a *trivial degeneration* of *gen* by

$$p' \;\stackrel{\text{def}}{=}\; \ulcorner \lambda(x, y).[\![sint]\!]_L\,[\,[\![gen]\!]_L\,x, y\,]\urcorner_L\ . \tag{9}$$

Program p' performs its computation in two stages: first, a specialized program is produced by *gen*, then the specialized program is evaluated by *sint*. For practical reasons, a trivial degeneration is not interesting, but it is sufficient to show that the computation of every generating extension can be unstaged by an interpreter.

The 2nd Futamura projection [5] asserts that for every interpreter there exists a translator $(\forall int\ \exists trans)$. From the above construction we see that for every translator, there exists an interpreter $(\forall trans \exists int)$. An $N \to L$-translator *trans* is an L-generating extension of an N/L-interpreter *int* since for all N-programs p and for all data d we have the functional equivalence required by Def. 4:

$$[\![int]\!]_L\,[\,p, d\,] \;=\; [\![\,[\![trans]\!]_L\,p\,]\!]_L\,d\ .$$

Remark. A more efficient degeneration may be obtained by applying a program composer to the composition of *sint* and *gen* in (9). An *L-composer* is an L-program such that, for all L-programs q, q' and for all data x, y:

$$[\![\,[\![komp]\!]_L\,[\,q, q'\,]\,]\!]_L\,[\,x, y\,] \;=\; [\![q]\!]_L\,[\,[\![q']\!]_L\,x, y\,]\ . \tag{10}$$

Then the *1st degeneration projection* [10] is defined by

$$p'' \;=\; [\![komp]\!]_L\,[\,sint, gen\,]\ .$$

Programs p' and p'' are functionally equivalent. A non-trivial composer may generate a more efficient composition by eliminating intermediate code generation, parsing, and other redundant operations between the programs. A method for automatic program composition is deforestation, another is supercompilation. Different ways of degenerating generating extensions were studied in [10].

6 Translation Universality

The question has been raised, *e.g.*, [5, 12, 16], whether target programs produced by the Futamura projections can be as efficient as those produced by traditional, optimizing translators. We give a precise answer to this fundamental question.

We examine whether, for every translator *trans*, there exists an interpreter *int* such that the target programs produced by the 1st Futamura projection are at least as fast as the target programs produced by the translator. More formally, given a specializer *spec*, we examine the question whether $\forall trans, \exists int, \forall p$:

$$[\![spec]\!]_L\,[\,int, \mathrm{SD}, p\,] \;\leq_{Time}\; [\![trans]\!]_L\,p\;. \tag{11}$$

We call this property *translation universality* of a specializer. We will show that every Jones-optimal specializer with static expression reduction is translation universal. Thus, under these conditions, translation by the Futamura projections is as powerful as translation by any translator. We will also show that a specializer that is not Jones-optimal is not translation universal.

Theorem 1 (Jopt-$\leq_{Time}$ sufficient for Trans-univ-$\leq_{Time}$). *For all specializers $spec \in Spec_L$ with static expression reduction, the following holds:*

$$\begin{gathered}\exists sint \in Sint_L :\; Jopt_{\leq_{Time}}(spec, sint)\\ \Longrightarrow\\ \forall trans \in Trans_{N\to L},\; \exists int \in Int_{N/L},\; \forall p \in P_N :\\ [\![spec]\!]_L\,[\,int, \mathrm{SD}, p\,] \;\leq_{Time}\; [\![trans]\!]_L\,p\;.\end{gathered}$$

Proof. We proceed in two steps. First, we show how to build for each translator *trans* an interpreter *int*; then we prove that the target programs produced by specializing *int* are at least as fast as those produced by *trans*.

1. For each $N \to L$-translator *trans*, define an N/L-interpreter *int* by

$$int \stackrel{\text{def}}{=} \ulcorner \underbrace{\lambda(p, d).[\![sint]\!]_L\,[\,[\![trans]\!]_L\,p, d\,]}_{\text{trivial degeneration}} \urcorner_L\;. \tag{12}$$

   Given an N-program p and data d, the interpreter *int* performs p's computation in two stages: first, *trans* translates p into a target program written in L; then the self-interpreter *sint* evaluates that program with input d. From Defs. 2 and 3 it follows that *int* is an N/L-interpreter since $\forall p \in P_N, \forall d \in D$:

$$[\![int]\!]_L\,[\,p, d\,] \;=\; [\![p]\!]_N\,d\;. \tag{13}$$

2. For each *trans*, let *int* be the interpreter defined in (12). Let p be an N-program. Then we obtain a target program p' by specializing *int* with respect to p (the 1st Futamura projection as discussed in Sect. 3):

$$p' \;=\; [\![spec]\!]_L\,[\,int, \mathrm{SD}, p\,]\;. \tag{14}$$

Since *spec* reduces static expressions and since *int* is of a form that suits Def. 7, we can rewrite (14) and obtain a program p''' as follows:

$$\begin{aligned} & [\![spec]\!]_L \, [\, sint, \mathrm{SD}, [\![trans]\!]_L \, p \,] && (15) \\ = \; & [\![spec]\!]_L \, [\, sint, \mathrm{SD}, p'' \,] && (16) \\ = \; & p''' \; . && (17) \end{aligned}$$

When we evaluate the application of *trans* in (15), we obtain (16) where $p'' = [\![trans]\!] \, p$. Then p''' in (17) is the result of specializing *sint* with respect to p''. According to Def. 7, program p' in (14) is identical to program p''' in (17): $p' = p'''$. We conclude from this identity, the specialization in (16) and property $Jopt_{\leq_{Time}}(spec, sint)$ that $p' \leq_{Time} p''$, and we have $[\![spec]\!]_L \, [\, int, \mathrm{SD}, p \,] \leq_{Time} [\![trans]\!]_L \, p$. This relation holds for any p. □

The interpreter in (12) performs the computation in two stages: the first stage performs a transformation (translation) of the source program and the second stage evaluates the resulting program with a self-interpreter. Since the specializer performs static expression reduction, any computation can be performed on a program p. Jones optimality of the specializer then guarantees that the performance of the transformed program is not degraded by specialization through the self-interpreter *sint*. (A similar idea was used in [8] to show the power of binding-time improvements.)

We now prove that Jones optimality is a necessary condition for the specializer to be translation universal.

Theorem 2 (Jopt-$\leq_{Time}$ necessary for Trans-univ-$\leq_{Time}$). *For all specializers spec $\in Spec_L$, the following holds:*

$$\begin{gathered} \exists sint \in Sint_L : \; Jopt_{\leq_{Time}}(spec, sint) \\ \Longleftarrow \\ \forall trans \in Trans_{N \to L}, \; \exists int \in Int_{N/L}, \; \forall p \in P_N : \\ [\![spec]\!]_L \, [\, int, \mathrm{SD}, p \,] \; \leq_{Time} \; [\![trans]\!]_L \, p \; . \end{gathered}$$

Proof. Assume that the *rhs* of the implication holds for all languages, in particular for languages $N = L$. Choose an $L \to L$-translator $trans_{id}$ such that $[\![trans_{id}]\!]_L \, p = p$ and let *sint* be the L/L-interpreter (= self-interpreter) that satisfies the *rhs* with respect to $trans_{id}$. Using the identity $[\![trans_{id}]\!]_L \, p = p$, the *rhs* of the implication can be simplified to

$$[\![spec]\!]_L \, [\, sint, \mathrm{SD}, p \,] \; \leq_{Time} \; p \; . \tag{18}$$

Since this relation holds for any p, we conclude from Def. 10 that *sint* is a self-interpreter for *spec* such that $Jopt_{\leq_{Time}}(spec, sint)$. □

We conclude that, in terms of target-program efficiency, the translation power of a specializer that is not Jones-optimal is *strictly weaker* than a Jones-optimal specializer. A specializer that is not Jones-optimal cannot always achieve the

target-program efficiency a translator can, no matter how hard we try to find a suitable interpreter. The theorems hold for all specializers regardless of their source languages, their specialization methods, or any other operational details.

The proof of Thm. 1 is rendered by a trivial degeneration in (12). This construction is sufficient for theoretical purposes, but not very efficient: it incorporates an entire translator and uses trivial degeneration. In practice, there exist more customized interpreters that can induce the same target-program effect in connection with a particular specializer. For example, it is known how to build interpreters that are *not* based on trivial degeneration, while letting offline partial evaluators perform optimizing translations [15, 22], deforestation or other supercompilation effects [9].

Given a particular specialization algorithm, only those fragments that are relevant for guiding that algorithm need to be considered in the design of a suitable interpreter. Fusing the self-interpreter with the translator in (12) by a program composer may eliminate redundant interface operations between the two programs. This may lead to a more 'natural' interpreter than the one used in our proof. There is room for more work on interpreter design for specialization.

The proof of Thm. 2 makes use of an identity translator $trans_{id}$. The theorem can also be proven using a *non-degrading self-translator*, that is, any $L \to L$-translator *trans* such that $\forall p \in P_L : [\![trans]\!]_L\, p \leq_{Time} p$. Conversely, if a specializer *spec* is not Jones-optimal, then it is not possible to self-translate programs without degrading the efficiency of some source programs: $\forall sint \in Sint_L, \exists p \in P_L$: $[\![spec]\!]_L\, [\, sint, \mathrm{SD}, p\,] \not\leq_{Time} p$. Thus, a specializer that is not Jones-optimal, always degrades the performance of some target programs, and we have a good reason for not calling such a specializer translation universal.

The theorems do not imply that Jones optimality is necessary for every form of translation. A simple example is the class of *trivial translators*, that is, any $N \to L$-translator such that, for some N/L-interpreter *int*, we have $\forall p \in P_L$: $[\![trans_{triv}]\!]_L\, p = \ulcorner \lambda d.[\![int]\!]_L\, [\, p, d\,] \urcorner_L$. To match a trivial translator, all that is needed is a trivial specializer, but this translation is hardly interesting in practice. To summarize, there exist translators whose translation effect can be matched by a specializer without Jones optimality, but these translators *degrade the performance* of their source programs such that the specializer does not need to remove all of the interpretive overhead.

7 Generating Extensions and Jones Optimality

What we presented in the previous section can be generalized in two ways. First, instead of using a time order, we may want to use other orders on programs, such as a size order. Second, instead of using translators, we may want to consider generating extensions. A translator is just a special case of a generating extension. It is a generating extension of an interpreter.

First, the idea of Jones optimality can be applied to other partial orders or equivalence relations on programs. Let $\leq_R$ be such a binary relation on programs, then we define a parameterized form of Jones-optimality and write $Jopt_{\leq_R}$. For

instance, the relation could be $\leq_{Size}$ or $\leq_{Time}$. Since the relation is reflexive, we have $Jopt_{\leq_R}(spec_{jser}, mysint)$ for the specializer $spec_{jser}$ defined in Sect. 4.

Definition 11 (Jopt-$\leq_R$). *Let $sint \in Sint_L$ be a self-interpreter, then a specializer $spec \in Spec_L$ is* Jones-optimal for *sint* with respect to $\leq_R$ *iff* $Jopt_{\leq_R}(spec, sint)$ *where*

$$Jopt_{\leq_R}(spec, sint) \stackrel{\text{def}}{=} \forall p \in P_L : [\![spec]\!]_L\,[\,sint, \mathrm{SD}, p\,] \leq_R p\ .$$

Second, instead of translation universality, we now study the question whether, for every generating extension *gen* of a program p, there exists a program $p' \in P_L^p$ such that all programs produced by specializing p' are in relation $\leq_R$ with the programs produced by p's generating extension. Given a specializer *spec*, we examine the question whether $\forall gen, \exists p', \forall x$:

$$[\![spec]\!]_L\,[\,p', \mathrm{SD}, x\,] \leq_R [\![gen]\!]_L\,x\ . \tag{19}$$

In analogy to Sect. 6, we call this property *generation universality (with respect to* $\leq_R$*)*. We generalize the two theorems by using $Jopt_{\leq_R}$ instead of $Jopt_{\leq_{Time}}$ and a generating extension *gen* instead of a translator *trans*.

Theorem 3 (Jopt-$\leq_R$ sufficient for Gen-univ-$\leq_R$). *For all specializers spec $\in Spec_L$ with static expression reduction, the following holds:*

$$\begin{array}{c} \exists sint \in Sint_L : Jopt_{\leq_R}(spec, sint) \\ \Longrightarrow \\ \forall gen \in Gen_L^p,\ \exists p' \in P_L^p,\ \forall x \in D : \\ [\![spec]\!]_L\,[\,p', \mathrm{SD}, x\,] \leq_R [\![gen]\!]_L\,x\ . \end{array}$$

Theorem 4 (Jopt-$\leq_R$ necessary for Gen-univ-$\leq_R$). *For all specializers spec $\in Spec_L$, the following holds:*

$$\begin{array}{c} \exists sint \in Sint_L : Jopt_{\leq_R}(spec, sint) \\ \Longleftarrow \\ \forall gen \in Gen_L^p,\ \exists p' \in P_L^p,\ \forall x \in D : \\ [\![spec]\!]_L\,[\,p', \mathrm{SD}, x\,] \leq_R [\![gen]\!]_L\,x\ . \end{array}$$

Proofs (outline). The proof of Thm. 3 follows the one of Thm. 1 except that it makes use of the trivial degeneration of *gen* in (9). The proof of Thm. 4 utilizes the fact that every translator is a generating extension ($Trans_{N\to L} = Gen_L^{int}$ where $int \in Int_{N/L}$) and then follows the argument for Thm. 2. □

Example. Obtaining a linear string matcher out of a quadratic string matcher is a classic example in program specialization ([6]; see also [2]). This is a challenging problem because it is known that dedicated program generators exist that can produce efficient, specialized string matchers. For example, the Knuth-Morris-Pratt algorithm yields string matchers whose time complexity is linear in the length of the text and whose size is linear in the length of the pattern.

Let us look at the generation of specialized string matchers in an abstract way. A generator of specialized string matchers, such as the Knuth-Morris-Pratt algorithm, can be viewed as a generating extension *matchgen* of a general string matcher *match*. Let *pat* be a pattern and *txt* be a text, then *match* checks whether *pat* occurs in *txt*. The generator *matchgen*, and the specialized string matchers it produces, perform the same computation as the general string matcher, but in two stages. This is described by

$$[\![match]\!]_L\,[\, pat, txt\,] \;=\; [\![[\![matchgen]\!]_L\, pat]\!]_L\, txt\ .$$

As we recall from Sect. 4, specializers with static expression reduction exist that satisfy $Jopt_{=_{Text}}$ (where equivalence relation $=_{Text}$ relates programs that are textually identical up to variable renaming). Clearly, $p =_{Text} q \Rightarrow (p =_{Time} q \wedge p =_{Size} q)$. Let *spec* be such a specializer. This guarantees that for each generator *matchgen* a program *match* exists such that *spec* produces specialized string matchers that have the same time and size complexity as those produced by the generator *matchgen*. More precisely, given a specializer *spec* with static expression reduction and $Jopt_{=_{Text}}(spec, sint)$, we have $\forall matchgen, \exists match, \forall pat$:

$$\begin{aligned} [\![spec]\!]_L\,[\, match, \mathrm{SD}, pat\,] &=_{Time} [\![matchgen]\!]_L\, pat\ , \\ [\![spec]\!]_L\,[\, match, \mathrm{SD}, pat\,] &=_{Size} [\![matchgen]\!]_L\, pat\ . \end{aligned}$$

This guarantees that for every specializer that satisfies the stated conditions, a string matcher *exists* such that the specialization of that matcher with respect to a pattern yields specialized string matchers of the desired time and size complexity. It does *not* tell us how 'naive' the string matcher *match* can be for a particular specialization method (*e.g.*, offline partial evaluation, supercompilation, partial deduction, generalized partial computation) or which binding-time improvements need to be applied to trigger the desired optimization. The degeneration construction used in the proof of Thm. 3 guarantees that for every *spec* a program *match* can be constructed that satisfies the stated conditions.

Also, the theorem does *not* tell us whether Jones optimality is needed for a particular specialization problem (*cf.* the discussion about simulating trivial translators at the end of Sect. 6). However, if a specializer is not Jones-optimal, then we know that the specializer is not generation universal and we cannot be sure to find the desired source program p' for every generating extension *gen*.

The practical challenge in making program specialization useful is to design specialization algorithms that are strong enough such that, *e.g.*, the string matcher *match* can be as simple as possible. This is the familiar tradeoff between improving the overall transformation by modifying the specializer, inserting an instrumented interpreter [7], or binding-time improving the source programs.

8 Related Work

The first version of optimality appeared in [12, Problem 3.8] where a specializer was called "strong enough" if program p and program $[\![spec]\!]\,[\, sint, \mathrm{SD}, p\,]$ are

"essentially the same". The definition of optimality used in Def. 10 appeared first in [13, p.650]; see also [14, Sect. 6.4]. These first studies were motivated by the problem of self-application. We applied the idea of self-interpreter specialization and obtained general statements about translation universality and generation universality of certain classes of specializers. In related work, the role of Jones optimality for universal binding-time improvements was investigated [8] and the idea of controlling the properties of residual programs by specializing suitable interpreters was studied [7, 9, 15, 22].

It was also found [20] that a specializer is 'weak' in a more general sense if it cannot overcome inherited limits and that these limits are best observed by specializing a self-interpreter. Since the power of a specializer can be judged in different ways, it was suggested [17] to use the term "Jones optimality". Recent studies [3, 24] focused on the problem of tag-elimination to achieve Jones optimality when specializing self-interpreters for strongly-typed languages.

9 Conclusion

We found that Jones optimality plays an important role in simulating translators and generating extensions by program specialization. We introduced a parameterized notion of Jones optimality and showed that, in principle, a Jones-optimal specializer with static expression reduction can always produce residual programs that match the size and time complexity of programs produced by an arbitrary generating extension. We say that such a specializer is generation universal. It is also reassuring to know that, in general, the method of translating programs by specializing interpreters is as powerful as translating programs by ordinary translators. In short, there is no theoretical limit that would make Futamura's method strictly weaker than ordinary translation.

We should reemphasize that our results imply neither that a specializer that is not Jones-optimal is useless nor that program generation by specialization is always a good idea. They show that there exists a theoretical limit to what can be achieved when a specializer is not generation universal. One of the main motivations for studying automatic program specialization is that a specializer has pragmatic success in producing efficient residual programs for a large class of source programs and, thus, a compromise between theory and practice needs to be found when designing a specializer.

We focused on the role of Jones optimality in specialization. We believe that the power to perform universal computations is another property necessary for generation universality. Thus, a total specializer can be said to be limited in its theoretical capability to simulate arbitrary translators and generating extensions. Whether the results in this paper can be adapted to other cases of non-standard interpreter hierarchies [1] is a topic for future work. We also did not explore the conditions for generating translators and generating extensions by specializing specializers (2nd Futamura projection). It is quite possible that the results in this paper can be carried to the next metasystem level.

Acknowledgments We would like to thank Sergei Abramov, Mikhail Bulyonkov, Yoshihiko Futamura, Masahiko Kawabe, and the anonymous reviewers for valuable comments on an earlier version of this paper.

References

1. S. M. Abramov, R. Glück. Combining semantics with non-standard interpreter hierarchies. In S. Kapoor, S. Prasad (eds.), *Foundations of Software Technology and Theoretical Computer Science. Proceedings*, LNCS 1974, 201–213. Springer-Verlag, 2000.
2. M. S. Ager, O. Danvy, H. K. Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 32–46. ACM Press, 2002.
3. O. Danvy, P. E. M. López. Tagging, encoding, and Jones optimality. In P. Degano (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2618, 335–347. Springer-Verlag, 2003.
4. A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.
5. 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.
6. Y. Futamura, K. Nogi. Generalized partial computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 133–151. North-Holland, 1988.
7. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, 1994.
8. R. Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 9–19. ACM Press, 2002.
9. R. Glück, J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier (ed.), *Static Analysis. Proceedings*, LNCS 864, 432–448. Springer-Verlag, 1994.
10. R. Glück, A. V. Klimov. A regeneration scheme for generating extensions. *Information Processing Letters*, 62(3):127–134, 1997.
11. J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43(2–3):149–167, 1986.
12. N. D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 1–14. North-Holland, 1988.
13. N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson (ed.), *Automata, Languages and Programming. Proceedings*, LNCS 443, 639–659. Springer-Verlag, 1990.
14. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
15. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Conference Record of the Nighteenth Symposium on Principles of Programming Languages*, 258–268. ACM Press, 1992.
16. S. S. Lavrov. On the essence of mixed computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 317–324. North-Holland, 1988.

17. H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha (ed.), *Semantics, Applications, and Implementation of Program Generation. Proceedings*, LNCS 1924, 129–148. Springer-Verlag, 2000.
18. T. Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 325–347. North-Holland, 1988.
19. T. Æ. Mogensen. Separating binding times in language specifications. In *Fourth International Conference on Functional Programming and Computer Architecture, London, UK*, 14–25. ACM Press and Addison-Wesley, Sept. 1989.
20. T. Æ. Mogensen. Evolution of partial evaluators: removing inherited limits. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 303–321. Springer-Verlag, 1996.
21. P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger, N. D. Jones (eds.), *Programs as Data Objects*, LNCS 217, 236–256. Springer-Verlag, 1985.
22. P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 446–464. Springer-Verlag, 1996.
23. S. Skalberg. Mechanical proof of the optimality of a partial evaluator. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, 1999.
24. W. Taha, H. Makholm, J. Hughes. Tag elimination and Jones-optimality. In O. Danvy, A. Filinsky (eds.), *Programs as Data Objects. Proceedings*, LNCS 2053, 257–275. Springer-Verlag, 2001.