Jones Optimality, Binding-Time Improvements, and the Strength of Program Specializers

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

ABSTRACT

Jones optimality tells us that a program specializer is strong enough to remove an entire level of self-interpretation. We show that Jones optimality, which was originally aimed at the Futamura projections, plays an important role in binding-time improvements. The main results show that, regardless of the binding-time improvements which we apply to a source program, no matter how extensively, a specializer that is not Jones-optimal is strictly weaker than a specializer which is Jones optimal. By viewing a binding-time improver as a generating extension of a self-interpreter, we can connect our results with previous work on the interpretive approach.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation*; D.3.4 [**Programming Languages**]: Processors—*interpreters, preprocessors*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms

Languages, Theory

Keywords

binding-time improvements, Futamura projections, interpretive approach, Jones optimality, metacomputation, self-interpreters, specializer projections

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

ASIA-PEPM'02, September 12-14, 2002, Aizu, Japan.

1. INTRODUCTION

Binding-time improvements are semantics-preserving transformations that are applied to a source program prior to program specialization. Instead of specializing the original program, the modified program is specialized. The goal is to produce residual programs that are better in some sense than the ones produced from the original program. A classical example [6] is the binding-time improvement of a naive pattern matcher so that an offline partial evaluator [20] can produce from it specialized pattern matchers that are as efficient as those generated by the Knuth, Morris & Pratt (KMP) algorithm (an offline partial evaluator cannot achieve this optimization without a suitable binding-time improvement).

It is well-known that two programs which are functionally equivalent may specialize very differently. Binding-time improvements can lead to faster residual programs by improving the flow of static data, or make a specializer terminate more often by dynamizing static computations. Numerous binding-time improvements are described in the literature (*e.g.*, [3, 7, 20]); they are routinely used for offline and online specializers. The main advantage is that they do not require a user to modify the specializer in order to overcome the limitations of its specialization method. Hence, they are handy in many practical situations.

Not surprisingly, several questions have been raised about binding-time improving programs: What are the limitations of binding-time improvements, and under what conditions can these modifications trigger a desired specialization effect? Is it good luck that a naive pattern matcher may be rewritten so as to lead an offline partial evaluator to perform the KMP-optimization, or can such binding-time improvements be found for any problem and for any specializer? This paper answers these and related questions on a conceptual level. We will not rely on a particular specialization method or on other technical details. We are interested in statements that are valid for all specializers, and we have identified such conditions.

Figure 1 shows the structure of a specializer system employing a binding-time improver as a preprocessor. The binding-time improver *bti* takes a program p and a division SD, which classifies p's parameters as static and dynamic, and returns a functionally equivalent program p'. This program is then specialized with respect to the static data x by the specializer *spec*. The specializer and the binding-time improver take the division SD as input, but the static

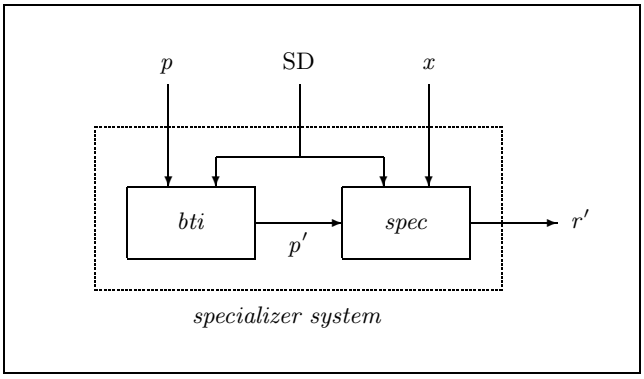

Figure 1: Binding-time improver as preprocessor of a program specializer

data x is only available to the specializer. This flow of the transformation is common to virtually all specializer systems. Usually, the specializer is an automatic program transformer, while the binding-time improvements are often done by hand. For our investigation, it does not matter how these steps are performed.

The results in this paper extend previous work on Jones optimality [18, 28, 22, 32] and the interpretive approach [11, 13, 34]. We will see that Jones optimality [18] plays a key role in the power of binding-time improvers and, together with static expression reduction, establishes a certain kind of non-triviality. Among others, we will precisely answer the old question whether an offline partial evaluator can be as powerful as an online partial evaluator.

This paper is organized as follows. After reviewing standard definitions of interpreters and specializers (Sect. 2), we present the two main ingredients for our work, Jones optimality (Sect. 3) and binding-time improvers (Sect. 4). The main results are proven (Sect. 5) and additional theorems are presented (Sect. 6). The connection with the interpretive approach is established (Sect. 7) and opportunities for optimizing our constructions are discussed (Sect. 8). We conclude with related work (Sect. 9) and challenges for future work (Sect. 10).

We assume that the reader is familiar with partial evaluation, *e.g.*, as presented in [20, Part II].

2. PRELIMINARIES

This section reviews standard definitions of interpreters and specializers. The notation is adapted from [20]; we use divisions (SD) when classifying the parameters of a program as static and dynamic.[1] We assume that we are always dealing with universal programming languages.

2.1 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 (when the index $_L$ is unspecified, we assume that a language L is intended). Multiple arguments are written as list, such as $[\![p]\!]_L\,[\,d_1, ..., d_n\,]$. The notation is strict in its arguments.

Equality between program applications shall always mean strong (computational) equivalence: either both sides of an equation are defined and equal, or both sides are undefined. Programs and their input and output are drawn from a common data domain D. Including all program texts in D is convenient when dealing with programs that accept both programs and data as input (a suitable choice for D is the set of lists known from Lisp). We define a program domain $P \subseteq D$, but leave the programming language unspecified. Elements of the data domain evaluate to themselves. Programs $p \in P$ are applied by enclosing them in $[\![\cdot]\!]$.

When we define a program using λ-abstraction, the expression needs to be translated into the corresponding programming language (denoted by $\ulcorner \lambda ... \urcorner \in P$). The translation is always possible when L is a universal programming language.

2.2 Interpreters and Specializers

What follows are standard definitions of interpreters and specializers.

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

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

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

Definition 3. (specializer) An L-program $spec \in Spec$ is a *specializer for L* iff $\forall p \in P, \forall x, y \in D$:

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

For simplicity, we assume that the programs which we specialize have two arguments, and that the first argument is static. Even though we make use of only division SD in the definition, we keep it explicit (for reasons explained in [12]). A specializer need not be total. The definition allows $spec$ to diverge on $[\,p, \mathrm{SD}, x\,]$ iff p diverges on $[\,x, y\,]$ for all y. In practice, specializers often sacrifice the termination behavior. For a discussion of termination issues see [19].

A specializer is *trivial* if the residual programs it produces are simple instantiations of the source programs.

Definition 4. (trivial specializer) An L-specializer $spec_{triv} \in Spec$ is *trivial* iff $\forall p \in P, \forall x, y \in D$:

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

More realistic specializers evaluate static expressions in a source program. An expression is static if it depends only on known data and, thus, can be precomputed at specialization time. We define a special case of static expression reduction which is sufficient for our purposes. The definition of running time is taken from [20, Sect. 3.1.7].[2]

Definition 5. (running time) For program $p \in P$ and data $d_1, ..., d_n \in D$, let $t_p(d_1, ..., d_n)$ denote the *running time* to compute $[\![p]\!]\,[\,d_1, ..., d_n\,]$.

Definition 6. (static expression reduction) A specializer $spec \in Spec$ has *static expression reduction* if $\forall q, q' \in P$, $\forall x, y \in D$:

$$\begin{array}{lcl} t_{r'}(y) & = & t_r(y) \\ \textit{where} \quad r' & = & [\![spec]\!]_L\,[\,p, \mathrm{SD}, x\,] \\ r & = & [\![spec]\!]_L\,[\,q, \mathrm{SD}, [\![q']\!]_L\, x\,] \\ p & \stackrel{\mathrm{def}}{=} & \ulcorner \lambda(a, b).[\![q]\!]_L\,[\,[\![q']\!]_L\, a, b\,]\urcorner \end{array}$$

[1] This does *not* imply that our specializers use offline partial evaluation.

[2] The measure for the running time can be a timed semantics (*e.g.*, the number of elementary computation steps).

The definition tells us that, in terms of residual-program efficiency, there is no difference between specializing program q with respect to a value $[\![q']\!]\,x$ and specializing the composition p of q and q' with respect to a value x. This implies that the specializer contains at least an interpreter (a universal program) to evaluate static applications (here $[\![q']\!]\,a$ in the definition of p).

A specializer with static expression reduction is non-trivial. The definition implies that it has the power to perform universal computations. Most specializers that have been implemented, including online and offline partial evaluators, try to evaluate as many static expressions as possible to improve the efficiency of the residual programs. They satisfy the static expression reduction property given above.

3. JONES OPTIMALITY

When translating a program by specializing an interpreter, it is important that the entire interpretation overhead is removed. Let us look at this problem more closely and then explain the definition of Jones optimality.

3.1 Translation by Specialization

Let p be an N-program, let $intN$ be an N/L-interpreter, and let $spec$ be a specializer for L. Then the *1st Futamura projection* [9] is defined by

$$q \;=\; [\![spec]\!]_L\,[\,intN, \mathrm{SD}, p\,]\;. \qquad (1)$$

Using Defs. 1 and 3 we have the functional equivalence between q and p:

$$[\![q]\!]_L\,d \;=\; [\![intN]\!]_L\,[\,p, d\,] \;=\; [\![p]\!]_N\,d\;. \qquad (2)$$

Note that program p is written in language N, while program q is written in language L. This N-to-L-translation was achieved by specializing the N/L-interpreter $intN$ with respect to p. We say that q is the target program of source program p. The translation can always be performed. Consider a trivial specializer. The first argument of $intN$ is instantiated to p, and the result is a trivial target program:

$$[\![spec_{triv}]\!]_L\,[\,intN, \mathrm{SD}, p\,] \;=\; \ulcorner \lambda d.[\![intN]\!]_L\,[\,p, d\,]\urcorner\;. \qquad (3)$$

Clearly, this is not the translation we expect. The target program is inefficient: it contains an entire interpreter. A natural goal is therefore to produce target programs that are as efficient as their source programs. Unfortunately, we cannot expect a specializer to produce efficient target programs from *any* interpreter. For non-trivial languages, no specializer exists [16] that could make 'maximal use' of the static input, here p, in all programs.

3.2 Jones-Optimal Specialization

A specializer is said to be "strong enough" [18] if it can completely remove the interpretation overhead of a self-interpreter. The definition is adapted from [20, 6.4]; it makes use of the 1st Futamura projection.

Definition 7. (Jones optimality) Let $sint \in Sint$ be a self-interpreter, then a specializer $spec \in Spec$ is *Jones-optimal for sint iff* $Jopt(spec, sint)$ where

$$Jopt(spec, sint) \stackrel{\mathrm{def}}{=} \left\{ \begin{array}{c} \forall p \in P,\ \forall d \in D: \\ p' = [\![spec]\!]\,[\,sint, \mathrm{SD}, p\,] \\ t_{p'}(d) \le t_p(d) \end{array} \right.$$

A specializer *spec* is said to be *Jones-optimal* if there exists at least one self-interpreter *sint* such that, for all source programs p, the target program p' is at least as efficient for all inputs d as the source program p. This tells us that *spec* can remove an entire layer of self-interpretation.

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 programs p and p' are identical, it is safe to conclude that this goal is achieved. Also, as explained in [25], the limits on the structure of residual programs that are inherited from structural bounds in the source programs are best observed by specializing a self-interpreter (*e.g.*, the arity of function definitions in a residual program).

It is easy to require Jones optimality, but it is not always easy to satisfy it. For instance, for a partial evaluator for a first-order functional language with algebraic data types [25], a combination of several transformation methods is necessary (constant folding, unfolding, polyvariant specialization, partially static values, constructor specialization, type specialization).

Jones optimality was first proven [28] for lambda-mix; see also [24]. The first implementation of a Jones-optimal specializer was the offline partial evaluator [26] for a Lisp-like language. An early offline partial evaluator [23] for a similar language utilizes partially static structures to produce near-identity mappings. For many specializers it is not known whether they are Jones-optimal or not. For other partial evaluators, such as FCL-mix [15], it is impossible to write a self-interpreter that makes them Jones optimal. Recent work [32] on Jones optimality concerns tag-elimination when specializing self-interpreters for strongly typed languages.

Note that the definition of Jones optimality can be satisfied by a simple construction

$$\begin{array}{l} myspec \stackrel{\mathrm{def}}{=} \ulcorner \lambda(p, \mathrm{SD}, x)\,.\,\mathit{if}\ equal(p, mysint)\ \mathit{then}\ x \\ \qquad\qquad\qquad\qquad \mathit{else}\ [\![spec_{triv}]\!]\,[\,p, \mathrm{SD}, x\,]\urcorner \end{array}$$

where *mysint* is a fixed self-interpreter for which we want *myspec* to be Jones-optimal and $spec_{triv}$ is the trivial specializer from Def. 4. Specializer *myspec* returns the last argument, x, unchanged if the first argument, p, is equal to *mysint*; otherwise, *myspec* performs a trivial specialization. Clearly, such a specializer is not useful in practice, but formally $Jopt(myspec, mysint)$.

Finally, we show that a Jones-optimal specializer with static expression reduction exists. This fact will play an important role in the next sections. Both properties can be satisfied by a simple construction

$$\begin{array}{l} myspec_{stat} \stackrel{\mathrm{def}}{=} \ulcorner \lambda(p, \mathrm{SD}, x)\,.\,\mathit{case}\ p\ \mathit{of} \\ \quad \text{`}\lambda(a, b).[\![q]\!]\,[\,[\![q']\!]\,a, b\,]\text{'} \rightarrow [\![myspec]\!]\,[\,q, \mathrm{SD}, [\![sint]\!]\,[\,q', x\,]\,] \\ \quad \mathit{otherwise} \qquad\qquad\quad \rightarrow [\![myspec]\!]\,[\,p, \mathrm{SD}, x\,]\urcorner \end{array}$$

where *case* implements pattern matching: if program p consists of a composition of two arbitrary programs, then p is decomposed[3] into q and q', and 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. The specializer is Jones-optimal for *mysint* and has the expression reduction property of Def. 6. More realistic

[3]We shall not be concerned with the technical details of parsing an L-program p.

Jones-optimal specializers, for instance [26, 23, 25, 28], also satisfy the property of static expression reduction.

4. BINDING-TIME IMPROVERS

Binding-time improvements [20] are semantics-preserving transformations that are applied to a source program before specialization with the aim to produce residual programs that are better in some sense than those produced from the original source program. Numerous binding-time improvements have been described in the literature (*e.g.*, [3, 7, 20]).

Formally, a *binding-time improver* is a program that takes a program p and a division of p's arguments into static and dynamic, and then transforms p into a functionally equivalent program p'. A binding-time improver performs the transformation independently of the static values that are available to a specializer.

Definition 8. (binding-time improver) An L-program $bti \in Bti$ is a *binding-time improver for* L iff $\forall p \in P,\ \forall x.y \in D$:

$$[\![bti]\!]_L\,[\,p, \mathrm{SD}\,] \;=\; p' \quad s.t. \quad [\![p']\!]_L\,[\,x, y\,] \;=\; [\![p]\!]_L\,[\,x, y\,]$$

Using Defs. 3 and 8, we have the following functional equivalence between the residual programs r and r' produced by directly specializing p and specializing a binding-time improved version of p:

$$\begin{array}{lrcl} [\![r]\!]_L\,y & = & [\![r']\!]_L\,y & \\ where & r & = & [\![spec]\!]_L\,[\,p, \mathrm{SD}, x\,] \\ & r' & = & [\![spec]\!]_L\,[\,[\![bti]\!]_L\,[\,p, \mathrm{SD}\,], \mathrm{SD}, x\,] \end{array} \tag{4}$$

In practice, we expect (and hope) that program r' is better, in some sense, than program r. For example, binding-time improvements can lead to faster residual programs by improving the flow of static information at specialization time.

It is important to note that a binding-timer improver cannot precompute a residual program since there are no static values, nor can it make a table containing all possible residual programs since in general the specialization of a source program allows for an infinite number of different residual programs. (Compare this to a translator: a translator cannot just precompute all results of a source program and store them, say, in a table in the target program since, in general, there is an infinite number of different results.)

The term "binding-time improvement" was originally coined in the context of offline partial evaluation [20] to improve the flow of static information, but such easing transformations are routinely used for all specialization methods. This is not surprising since there exists no specialization method that could make 'maximal use' of available information in all cases. We use the term binding-time improvement for any semantics-preserving pre-transformation and regardless of a particular specialization method.

Note that Def. 8 does not require that each source program is transformed: the identity transformation,

$$[\![bti_{id}]\!]\,[\,p, \mathrm{SD}\,] \quad = \quad p \tag{5}$$

is a correct (but trivial) binding-time improver. Of course, for a binding-time improver to be useful, it must perform non-trivial transformations at least on some programs; otherwise, the residual programs r and r' in (4) are always identical.

5. JONES OPTIMALITY & BINDING-TIME IMPROVEMENTS

In the previous sections, two different streams of ideas were presented, binding-time improvements and Jones optimality for the specialization of programs. How are they related? We put these ideas together and present the two main results.

5.1 Sufficient Condition

The first theorem tells us that for every Jones-optimal specializer $spec_1$ there exists a binding-time improver that allows the specializer to achieve the residual-program efficiency of any other specializer (Jones optimality is a sufficient condition). The proof makes use of a general construction of such a *bti*.

THEOREM 1. *(Jones optimality is sufficient) For all specializers $spec_1 \in Spec$ with static expression reduction, the following holds:*

$$\begin{array}{c} \exists sint_1 \in Sint :\ Jopt(spec_1, sint_1) \\ \Longrightarrow \\ \forall spec_2 \in Spec,\ \exists bti \in Bti,\ \forall p \in P,\ \forall x, y \in D : \\ r' \;=\; [\![spec_1]\!]\,[\,[\![bti]\!]\,[\,p, SD], SD, x\,] \\ r \;=\; [\![spec_2]\!]\,[\,p, SD, x\,] \\ t_{r'}(y) \le t_r(y)\ . \end{array}$$

Proof: We proceed in two steps.

1. Let $sint_1$ be the self-interpreter for which $spec_1$ is Jones-optimal. For each specializer $spec_2$, define a binding-time improver bti:

$$[\![bti]\!]\,[\,p, \mathrm{SD}\,] \stackrel{\mathrm{def}}{=} \ulcorner \lambda(x, y).\underbrace{[\![sint_1]\!]\,[\,[\![spec_2]\!]\,[\,p, \mathrm{SD}, x\,], y\,]}_{\text{two stages}}\urcorner\ . \tag{6}$$

   The binding-time improver depends on $sint_1$ and $spec_2$, but not on program p. Given a program p and a division SD, program bti produces a new program that performs p's computation in two stages: first, $spec_2$ specializes p with respect to x, then $sint_1$ evaluates the residual program with the remaining input y. From Defs. 2 and 3, it follows that the new program is functionally equivalent to p. According to Def. 8, bti is a binding-time improver since $\forall p \in P, \forall x, y \in D$:

$$[\![p]\!]\,[\,x, y\,] \;=\; [\![p']\!]\,[\,x, y\,] \quad where \quad p' \;=\; [\![bti]\!]\,[\,p, \mathrm{SD}\,]\ .$$

2. Consider the *rhs* of the implication in Thm. 1. For each $spec_2$, let bti be the binding-time improver defined in (6). Let p be a program and let x be some data, then we have the binding-time improved program

$$\begin{array}{rcl} p' & = & [\![bti]\!]\,[\,p, \mathrm{SD}\,] \\ & = & \ulcorner \lambda(x, y).[\![sint_1]\!]\,[\,[\![spec_2]\!]\,[\,p, \mathrm{SD}, x\,],\ y\,]\urcorner \end{array} \tag{7}$$

   and obtain the residual program r' by specializing p' with respect to x:

$$r' \quad = \quad [\![spec_1]\!]\,[\,p', \mathrm{SD}, x\,] \tag{8}$$

   Since $spec_1$ reduces static expressions and since p' is of a form that suits Def. 6, we can rewrite (8) as (9) and

obtain a program r'':

$$\begin{aligned} & [\![spec_1]\!]\,[\,sint_1, \mathrm{SD}, [\![spec_2]\!]\,[\,p, \mathrm{SD}, x\,]\,] && (9)\\ = \;& [\![spec_1]\!]\,[\,sint_1, \mathrm{SD}, r\,] && (10)\\ = \;& r''\ . && (11)\end{aligned}$$

After evaluating the application of $spec_2$ in (9), we obtain (10); recall that $r = [\![spec_2]\!]\,[\,p, \mathrm{SD}, x\,]$ in the *rhs* of the implication. Then r'' in (11) is the result of specializing $sint_1$ with respect to r by $spec_1$. Program r' is as fast as r'' according to Def. 6. From the specialization in (10) and $Jopt(spec_1, sint_1)$, we conclude that r' is at least as fast as r since $\forall y \in D$:

$$t_{r'}(y) \le t_r(y)\ . \qquad (12)$$

This relation holds for any p and for any x. This proves the theorem.

∎

5.2 Necessary Condition

The second theorem tells us that a specializer $spec_1$ that is not Jones-optimal cannot always reach the residual-program efficiency of another specializer $spec_2$ (Jones optimality is a necessary condition).

THEOREM 2. *(Jones optimality is necessary) For all specializers $spec_1 \in Spec$, the following holds:*

$$\begin{gathered} \exists sint_1 \in Sint :\ Jopt(spec_1, sint_1) \\ \Longleftarrow \\ \forall spec_2 \in Spec,\ \exists bti \in Bti,\ \forall p \in P,\ \forall x, y \in D : \\ r' \ = \ [\![spec_1]\!]\,[\,[\![bti]\!]\,[\,p, SD], SD, x\,] \\ r \ = \ [\![spec_2]\!]\,[\,p, SD, x\,] \\ t_{r'}(y) \le t_r(y)\ . \end{gathered}$$

Proof: Assume that the *rhs* of the implication holds. Choose $spec_2 \in Spec$ and $sint_2 \in Sint$ such that $Jopt(spec_2, sint_2)$. Such a specializer exists for any non-trivial programming language (*e.g.*, *myspec* in Sect. 3). Let *bti* be a binding-time improver which satisfies the *rhs* of the implication with respect to $spec_2$, and define

$$sint_1 \ = \ [\![bti]\!]\,[\,sint_2, \mathrm{SD}\,]\ . \qquad (13)$$

Since *bti* is a binding-time improver and $sint_2$ is a self-interpreter, $sint_1$ is also a self-interpreter. In the *rhs* of the implication, let p be $sint_2$, let x be a program q, and let y be some data d. Then we have

$$t_{r'}(d) \le t_r(d) \qquad (14)$$

where

$$\begin{aligned} r' \ &= \ [\![spec_1]\!]\,[\,sint_1, \mathrm{SD}, q\,] \\ r \ &= \ [\![spec_2]\!]\,[\,sint_2, \mathrm{SD}, q\,]\ . \end{aligned} \qquad (15)$$

Since $Jopt(spec_2, sint_2)$, we have that

$$t_r(d) \le t_q(d)\ . \qquad (16)$$

By combining (14) and (16), we realize that

$$t_{r'}(d) \le t_q(d)\ . \qquad (17)$$

This relation holds for any q and for any d, thus we conclude that $Jopt(spec_1,\ sint_1)$. This proves the theorem.

∎

The second theorem does not imply that a specializer that is not Jones-optimal cannot benefit from binding-time improvements, but that there is a limit to what can be achieved by a binding-time improver if $spec_1$ is not Jones-optimal.

Observe from the proof of Thm. 2 that the *rhs* of the implication can be weakened: instead of quantification "$\forall spec_2 \in Spec$", all that is needed is quantification "$\exists spec_2 \in Spec$, $\exists sint_2 \in Sint$, $Jopt(spec_2, sint_2)$".

Remark: In the literature [22], Jones optimality is often defined using $r' = r$ instead of $t_{r'}(y) \le t_r(y)$. Under this assumption, it follows directly from the *rhs* in Thm. 2 that $spec_1$ must reduce static expressions if $spec_2$ does.

5.3 Discussion

1. By combining both theorems, we can conclude that, in terms of residual-program efficiency, a specializer that is not Jones-optimal is *strictly weaker* than a specializer that is Jones-optimal.

2. The question whether an offline partial evaluator can be as powerful as an online partial evaluator can now be answered precisely: an offline partial evaluator with static expression reduction that is Jones-optimal can achieve the residual-program efficiency of any online partial evaluator by using a suitable binding-time improver. The binding-time improver depends on the partial evaluators but not on the source programs.

3. Jones optimality is important for more than just building specializers that work well with the Futamura projections. Previously it was found only in the intuition that it would be a good property [17, 18]. The theorems give formal status to the term "optimal" in the name of that criterion.

4. The results also support the observation [25] that a specializer has a weakness if it cannot overcome inherited limits and that they are best observed through specializing a self-interpreter (which amounts to testing whether a specializer is Jones-optimal).

A way to test the strength of a specializer is to see whether it can derive certain well-known efficient programs from naive and inefficient programs. One of the most popular tests [10, 6, 29, 2] is to see whether the specializer generates, from a naive pattern matcher and a fixed pattern, an efficient pattern matcher. What makes Jones optimality stand out in comparison to such tests is that while a Jones-optimal specializer with static expression reduction is guaranteed to pass *any* of these tests by a suitable binding-time improvement (Thm. 1), a specializer may successfully pass any number of these tests, but as long as it is not Jones-optimal, its strength is limited in some way (Thm. 2).

Even though the construction of the binding-time improver used in the proof of Thm. 1 suggests that each source program p is transformed into a new p', such a deep transformation may not be necessary in all cases. To what extent each source program needs to be transformed in practice depends on the desired optimization and the actual power of the specializer $spec_1$. More realistic binding-time improvers will not need to transform each source program.

6. ROBUSTNESS

This section presents two results regarding Jones optimality. They establish a certain kind of non-triviality for a Jones-optimal specializers with static expression reduction. In particular, they tell us that there is an *infinite number of self-interpreters* for which a Jones-optimal specializer with static expression reductions is also Jones-optimal.

THEOREM 3. *(Jones optimality not singularity) Let* $spec_1$, $spec_2 \in Spec$ *be two Jones-optimal specializers where* $spec_1$ *reduces static expressions, and let* $sint_1, sint_2 \in Sint$ *be two self-interpreters such that* $Jopt(spec_1, sint_1)$ *and* $Jopt(spec_2, sint_2)$*, then there exists a self-interpreter* $sint_3 \in Sint$*, different beyond renaming from* $sint_1$ *and* $sint_2$*, such that* $Jopt(spec_1, sint_3)$.

Proof: The proof uses a construction that combines two self-interpreters and a specializer into a new self-interpreter without breaking Jones optimality. Define the self-interpreter $sint_3$ by

$$sint_3 \stackrel{\text{def}}{=} \ulcorner \lambda(p,d).[\![sint_1]\!]\,[\,[\![spec_2]\!]\,[\,sint_2, \text{SD}, p\,], d\,]\urcorner\ . \quad (18)$$

The new self-interpreter $sint_3$ is different beyond renaming from the self-interpreters $sint_1$ and $sint_2$ since $sint_3$ contains both programs. To show $Jopt(spec_1, sint_3)$, we examine how $spec_1$ specializes $sint_3$. Since $spec_1$ reduces static expressions and since $sint_3$ is of a form that suits Def. 6, we have

$$\begin{aligned} & [\![spec_1]\!]\,[\,sint_3, \text{SD}, p\,] && (19)\\ = \ & [\![spec_1]\!]\,[\,sint_1, \text{SD}, [\![spec_2]\!]\,[\,sint_2, \text{SD}, p\,]\,] && (20)\\ = \ & [\![spec_1]\!]\,[\,sint_1, \text{SD}, p'\,] && (21)\\ = \ & p'' && (22)\end{aligned}$$

Let $p' = [\![spec_2]\!]\,[\,sint_2, \text{SD}, p\,]$, then we can conclude from $Jopt(spec_1, sint_1)$ and $Jopt(spec_2, sint_2)$ that

$$t_{p''}(d) \le t_{p'}(d) \le t_p(d)\ . \quad (23)$$

Since this relation holds for any p and any d, we have the desired property $Jopt(spec_1, sint_3)$. This proves the theorem. ∎

First, we observe that $spec_1$ can be 'more' Jones-optimal for $sint_3$ than $spec_2$ for $sint_2$: program p'' can be faster than p' which in turn can be faster than p. This is not surprising since specializers are usually not idempotent, and repeatedly applying a specializer to a program can lead to further optimizations. This is known from the area of compiler construction where an optimization may enable further optimization. This also underlines that it is realistic to choose the timing condition ($\le$) in Def. 7, as already discussed in Sect. 3.

Second, as a special case of Thm. 3, let $spec_1 = spec_2$ and $sint_1 = sint_2$. Then we can conclude by repeatedly applying the theorem that for every Jones-optimal specializer with static expression reduction there exists an infinite number of self-interpreters for which the specializer is also Jones optimal. We state this more formally in the following theorem.

THEOREM 4. *(Jones optimality is robust) Let* $spec \in Spec$ *be a specializer with static expression reduction and let* $sint \in Sint$ *be a self-interpreter such that* $Jopt(spec, sint)$*, then we have:*

1. *There exists an infinite number of self-interpreters* $sint_i \in Sint$*, which are pairwise different beyond renaming, such that* $Jopt(spec, sint_i)$.
2. *There exists an infinite number of specializers* $spec_i \in Spec$*, which are pairwise different beyond renaming, such that* $Jopt(spec_i, sint)$.

Proof: Let $copy_0$ be a program such that $d = [\![copy_0]\!]\,d$ for all $d \in D$. Define any number of programs $copy_{i+1}$ as follows ($i \ge 0$):

$$copy_{i+1} \stackrel{\text{def}}{=} \ulcorner \lambda p.[\![copy_i]\!]\,([\![copy_0]\!]\,p)\urcorner\ . \quad (24)$$

For all $d \in D$, we have $d = [\![copy_{i+1}]\!]\,d$, and $copy_{i+1}$ and $copy_j$ are different beyond renaming ($i \ge j \ge 0$).

1. Define any number of programs $sint_i$ ($i \ge 0$):

$$sint_i \stackrel{\text{def}}{=} \ulcorner \lambda(p,d).[\![sint]\!]\,[\,[\![copy_i]\!]\,p, d\,]\urcorner\ . \quad (25)$$

   Each $sint_i$ is a self-interpreter, and all self-interpreters are different beyond renaming since they contain different copy programs. Each $sint_i$ is of the form used in Def. 6 and we conclude $Jopt(spec, sint_i)$ since

$$[\![spec]\!]\,[\,sint_i, \text{SD}, p\,] \ = \ [\![spec]\!]\,[\,sint, \text{SD}, p\,]\ . \quad (26)$$

2. Define any number of programs $spec_i$ ($i \ge 0$):

$$spec_i \stackrel{\text{def}}{=} \ulcorner \lambda(p, \text{SD}, d).[\![spec]\!]\,[\,[\![copy_i]\!]\,p, \text{SD}, d\,]\urcorner \quad (27)$$

   Each $spec_i$ is a specializer, and all specializers are different beyond renaming since they contain different copy programs. We conclude that $Jopt(spec_i, sint)$ since

$$[\![spec_i]\!]\,[\,sint, \text{SD}, p\,] \ = \ [\![spec]\!]\,[\,sint, \text{SD}, p\,]\ . \quad (28)$$

This proves the theorem. Remark: the first item can also be proven by using the construction in the proof of Thm. 3. ∎

The requirement for static expression reduction in both theorems hints at a certain kind of non-triviality of such Jones-optimal specializers (also suggested by Thm. 1). With static expression reduction, we can be sure that Jones optimality is not just a 'singularity' for a specializer, but that there is an infinite number of such self-interpreters. A Jones-optimal specializer with static expression reduction can be said to be *robust* with respect to certain non-trivial modifications of the self-interpreters. This effectively excludes the Jones-optimal specializer *myspec* defined in Sect. 3. This may be the kind of *fairness* sought for in earlier work [22].

7. THREE TECHNIQUES FOR BINDING-TIME IMPROVING PROGRAMS

In this section we establish the connection between three known techniques for doing binding-time improvements (Figure 2): using a stand-alone binding-time improver *bti* (c), as we did in the previous sections, and stacking a source program p with an instrumented self-interpreter (a, b). We show that Thm. 1 and Thm. 2 cover the two new cases after a few minor adaptions, and that all three techniques can produce the same residual programs. Thus, they are equally powerful with respect to doing binding-time improvements.

Consider the following fragments from Thm. 1 and Thm. 2 where p', the binding-time improved version of p, is specialized with respect to x:

$$p' = [\![bti]\!]\,[\,p, \mathrm{SD}\,] \quad (29)$$
$$r' = [\![spec_1]\!]\,[\,p', \mathrm{SD}, x\,]\,. \quad (30)$$

Observe that *bti* has the functionality of a translator—if we disregard the extra argument SD. It is well-known [20] that the *generating extension* [8] of an interpreter is a translator. Since a binding-time improver has the functionality of an L-to-L translator, it is a generating extension of a self-interpreter for L.

Instead of binding-time improving p by *bti*, we can specialize a suitable self-interpreter *sint* with respect to p. There are two ways to produce a residual program by specializing a self-interpreter: either directly by the specializer projections [11] (a) or incrementally by the Futamura projections [9] (b). We will now examine these two cases.

7.1 Incremental Specialization

Let *bti* be a binding-time improver, and suppose that $spec_0$ is a specializer and $sint_0$ a self-interpreter such that[4]

$$[\![spec_0]\!]\,[\,sint_0, \mathrm{SD}, p\,] = [\![bti]\!]\,[\,p, \mathrm{SD}\,]\,. \quad (31)$$

Using this equality, we obtain a new set of equations from (29, 30):

$$p' = [\![spec_0]\!]\,[\,sint_0, \mathrm{SD}, p\,] \quad (32)$$
$$r' = [\![spec_1]\!]\,[\,p', \mathrm{SD}, x\,]\,. \quad (33)$$

Thus, every binding-time improvement can be performed in a *translative way*, as in (29), or in an *interpretive way*, as in (32). An example [34] for the latter case is the polyvariant expansion of a source program by specializing a suitable self-interpreter, and then applying a binding-time monovariant offline specializer to the modified program. The overall effect of the transformation is that of a binding-time polyvariant offline specializer, even though only a binding-time monovariant offline specializer was used to produce the residual program. Similarly, it is known [21, 27, 30, 31] that optimizing translators can be generated from suitable interpreters. Such techniques can also be used in self-interpreters to improve the specialization of programs.

Theorems 1 and 2 carry over to the interpretive case, provided we replace in the *rhs* of their implication the quantification "$\exists bti \in Bti$" by the quantification "$\exists spec_0 \in Spec$, $\exists sint_0 \in Sint$" and use (32, 33) instead of (29, 30).

7.2 Direct Specialization

The two steps (32, 33) can also be carried out in one step. For notational convenience, let us first redefine the format of a self-interpreter and of a specializer. This will make it easier to accommodate programs with two and three arguments.

A self-interpreter $sint'$ for interpreting programs with two arguments can be defined by

$$[\![sint']\!]\,[\,p, x, y\,] = [\![p]\!]\,[\,x, y\,] \quad (34)$$

and a specializer $spec'$ for specializing programs with three arguments can be defined by

$$[\![[\![spec']\!]\,[\,q, \mathrm{SDD}, a\,]]\!]\,[\,b, c\,] = [\![q]\!]\,[\,a, b, c\,] \quad (35)$$
$$[\![[\![spec']\!]\,[\,q, \mathrm{SSD}, a, b\,]]\!]\,c = [\![q]\!]\,[\,a, b, c\,] \quad (36)$$

where program q is specialized with respect to one static argument, a, in (35) and with respect to two static arguments, a and b, in (36). The residual program then takes the remaining arguments as input. Note that each case has a different division.

Using $spec'$, the two steps (32, 33) can be carried out in one step. Let q be the self-interpreter $sint'$ defined above, let a be a two-argument program p, and let b and c be some arguments x and y, respectively, then p can be specialized with respect to x via $sint'$ using (36):

$$r'' = [\![spec']\!]\,[\,sint', \mathrm{SSD}, p, x\,]\,. \quad (37)$$

That r'' is a residual program of p follows from (34, 36):

$$[\![r'']\!]\,y = [\![sint']\!]\,[\,p, x, y\,] = [\![p]\!]\,[\,x, y\,]\,. \quad (38)$$

Let *bti* be a binding-time improver and let $spec_1$ be a specializer as in (29, 30), and suppose that $spec'$ is a specializer and $sint'$ a self-interpreter such that[5]

$$[\![spec']\!]\,[\,sint', \mathrm{SSD}, p, x\,] = [\![spec_1]\!]\,[\,[\![bti]\!]\,[\,p, \mathrm{SD}\,], \mathrm{SD}, x\,]\,. \quad (39)$$

Equation (37) is the 1st specializer projection [11] but for a self-interpreter instead of an interpreter. Specializing a program via an interpreter is also known as the *interpretive approach* [11]. An example [13] is the generation of a KMP-style pattern matcher from a naive pattern matcher by inserting an instrumented self-interpreter between the naive pattern matcher and an offline partial evaluator (Similix). As mentioned in Sect. 1, an offline partial evaluator cannot achieve this optimization, but specializing the naive matcher via an instrumented self-interpreter does the job. Another example is the simulation of an online partial evaluator by an offline partial evaluator [33], and the bootstrapping of other program transformers [30].

Note that the transformation in (37) can be optimized at another level: specializing $spec'$ with respect to $sint'$ yields a new specializer. This internalizes the techniques of the self-interpreter in the new specializer. This is know as the 2nd specializer projection [11]; results on generating optimizing specializers are reported elsewhere (*e.g.*, [13, 33, 30]).

Again, both main theorems apply to this case by replacing quantification "$\exists bti \in Bti$" by "$\exists sint' \in Sint$" on the *rhs* of the implication.

7.3 Summary

The three techniques for binding-time improving programs are summarized in Fig. 2. They can produce the same residual programs and, thus, are equally powerful with respect to binding-time improving programs. Which technique is preferable in practice, depends on the application at hand. There exist example applications in the literature for each case. The results in this section shed new light on the relation between binding-time improvements and the interpretive approach. Our two main theorems cover all three cases.

[4] For every *bti* there exists a pair ($spec_0$, $sint_0$) such that (31), and vice versa (Prop. 1 and 2 in Appendix A).

[5] For every pair ($spec_1$, *bti*) there exists a pair ($spec'$, $sint'$) such that (39), and vice versa (Prop. 3 and 4 in Appendix A).

Direct specialization:

a. One step (1st specializer projection):

$$r'' \;=\; [\![spec']\!]\,[\,sint', \mathrm{SSD}, p, x\,] \qquad (37)$$

Incremental specialization:

b. Two steps (1st Futamura projection):

$$p' \;=\; [\![spec_0]\!]\,[\,sint_0, \mathrm{SD}, p\,] \qquad (32)$$
$$r' \;=\; [\![spec_1]\!]\,[\,p', \mathrm{SD}, x\,] \qquad (33)$$

c. Three steps (2nd Futamura projection):

$$bti' \;=\; [\![spec']\!]\,[\,spec_0, \mathrm{SDD}, sint_0\,] \qquad (40)$$
$$p' \;=\; [\![bti']\!]\,[\,\mathrm{SD}, p\,] \qquad (41)$$
$$r' \;=\; [\![spec_1]\!]\,[\,p', \mathrm{SD}, x\,] \qquad (42)$$

Figure 2: Three techniques for binding-time improving programs

When we add to (29, 30) a step that generates the binding-time improver from a self-interpreter, we obtain (40, 41, 42). This is the 2nd Futamura projection. Detail: For formal reasons, due to the definition of specializer *spec'* in (35), the arguments of bti' are reordered.

8. OPTIMIZING THE THEORETICAL CONSTRUCTIONS

The proof of Thm. 1 makes use of a general binding-time improver. In many cases, only certain fragments that are relevant to binding-time improving programs for a particular specializer $spec_1$ need to be incorporated in a binding-time improver, not the entire specializer (here $spec_2$). In the remainder of this section, we will point out some possibilities for optimizing the general construction by further program specialization and program composition.

1. Specialization: We observe that the binding-time improved program p' produced by *bti* in (7) contains a specializer $spec_2$ whose argument is fixed to p. This makes the program structure of p' much too general. Usually, not all components of $spec_2$ are needed to specialize p. We can improve the construction by replacing $spec_2$ by a generating extension gen_p of p, a program which is specified by

$$[\![gen_p]\!]\, x \;=\; [\![spec_2]\!]\,[\,p, \mathrm{SD}, x\,]\;. \qquad (43)$$

A generating extension gen_p is a generator of p's residual programs. We can use the following program instead of p':

$$p'' \;\stackrel{\mathrm{def}}{=}\; \ulcorner \lambda(x,y).\underbrace{[\![sint_1]\!]\,[\,[\![gen_p]\!]\, x, y\,]}_{\text{composition}} \urcorner\;. \qquad (44)$$

2. Composition: We notice the fixed composition of $sint_1$ and gen_p in (44). An intermediate data structure, a residual program, is produced by gen_p and then consumed by $sint_1$. Methods for program composition may fuse the two programs and eliminate intermediate data structures, code generation, parsing and other redundant operations.

It will be interesting to examine whether some of the known binding-time improvements can be justified starting from the general construction, and whether new binding-time improvers can be derived from the general construction where $spec_2$ represents the desired level of specialization.

Lets us illustrate this with an example. Suppose *onpe* is an online partial evaluator and *offpe* is an offline partial evaluator which reduces static expressions and is Jones-optimal for self-interpreter *sint*. We have the specializers

$$[\![onpe]\!]\,[\,p, \mathrm{SD}, x\,] \;=\; r \quad s.t. \quad [\![r]\!]\; y \;=\; [\![p]\!]\,[\,x, y\,]$$
$$[\![offpe]\!]\,[\,p, \mathrm{SD}, x\,] \;=\; r' \quad s.t. \quad [\![r']\!]\, y \;=\; [\![p]\!]\,[\,x, y\,]\;.$$

According to Thm. 1 we have $t_{r''}(y) \le t_r(y)$ where the two residual programs are produced by

$$r \;= [\![onpe]\!]\,[\,p, \mathrm{SD}, x\,]$$
$$r'' = [\![offpe]\!]\,[\,[\![bti]\!]\,[\,p, \mathrm{SD}\,], \mathrm{SD}, x\,]$$

and a binding-time improver defined by

$$[\![bti]\!]\,[\,p, \mathrm{SD}\,] \;\stackrel{\mathrm{def}}{=}\; \ulcorner \lambda(x,y)\,.\,[\![sint]\!]\,[\,[\![onpe]\!]\,[\,p, \mathrm{SD}, x\,], y\,] \urcorner\;.$$

After binding-time improving p with *bti*, the performance of r'' produced by *offpe* is at least as good as the one of r produced by *onpe*. Can we derive a useful binding-time improver from *bti* by program specialization and program composition? Can we obtain automatically one of the binding-time improved programs published in the literature (*e.g.*, [6, 2]) by specializing *bti* with respect to a source program (*e.g.*, a naive pattern matcher)?

Another challenging question is whether a binding-time improver can be specified by combining 'atomic' binding-time improvements, and how this relates to combining semantics by towers of non-standard interpreters [1].

9. RELATED WORK

The first version of optimality appeared in [17, Problem 3.8] where a specializer was called "strong enough" if program p and program $[\![spec]\!]\,[\,sint, \mathrm{SD}, p\,]$ are "essentially the same programs". The definition of optimality used in this paper appeared first in [18, p.650]; see also [20, Sect. 6.4]. Since the power of a specializer can be judged in many different ways, we use the term "Jones optimality" as proposed in [22]. These works focus on the problem of self-application; none of them considers the role of Jones-optimal specialization for binding-time improvements. Also, it was said [25] that a specializer is 'weak' if it cannot overcome inherited limits and that this is best observed by specializing a self-interpreter. Our results underline this argument from the perspective of binding-time improvements.

The term "binding-time improvement" was originally coined in the context of offline partial evaluation [20], but source-level transformations of programs prior to specialization are routinely used in connection with any specialization method. Binding-time improvements range from rearrangements of expressions (*e.g.*, using the associativity of arithmetic operations) to transformations which require some ingenuity (*e.g.*, the transformation of naive pattern matchers [6, 2]). Some transformations incorporate fragments of

a specializer [4]. A collection of binding-time improvements for offline partial evaluation can be found in [3, Sect.7] and in [20, Chapter 12]. The actual power of binding-time improvements was not investigated in these works.

The idea [11] of controlling the properties of residual programs by specializing a suitable interpreters was used to perform deforestation and unification-based information propagation [13, 14], binding-time polyvariant specialization [34, 30], and other transformations [30] with an offline partial evaluator. These apparently different streams of work are now connected to binding-time improvements.

Related work [5] has shown that, without using binding-time improvements, an offline partial evaluator with a maximally polyvariant binding-time analysis is functionally equivalent to an online partial evaluator, where both partial evaluators are based on constant propagation. This paper shows that any offline partial evaluator can simulate any online partial evaluator with a suitable binding-time improver, provided the offline partial evaluator is Jones-optimal.

10. CONCLUSION

Anecdotal evidence suggests that the binding-time improvement of programs is a useful and inexpensive method to overcome the limitations of a specializer without having to modify the specializer itself. These pre-transformation are often ad hoc, and applied to improve the specialization of a few programs. It was not know what the theoretical limitation and conditions for this method are, and how powerful such pre-transformation can be.

Our results show that one can always overcome the limitations of a Jones-optimal specializer that has static expression reduction by using a suitable binding-time improvement, and that this is not always the case for a specializer that is not Jones-optimal. Thus, regardless of the binding-time improvements which we perform on a source program, no matter how extensively, a specializer which is not Jones-optimal can be said to be strictly weaker than a specializer which is Jones-optimal. This also answers the question whether an offline partial evaluator can be as powerful as an online partial evaluator.

Jones optimality was originally formulated to assess the quality of a specializer for translating programs by specializing interpreters. Our results give formal status to the term "optimal" in the name of that criterion. Previously it was found only in the intuition that it would be a good property; indeed, it was proposed to rename it from "optimal specialization" to "Jones-optimal specialization" precisely because it was felt to be wrong to imply that no specializer can be better than a Jones-optimal one. This paper shows a way in which this implication is indeed true.

The proofs make use of a construction of a binding-time improver that works for any specializer. This construction is sufficient for theoretical purposes, but not very practical (it incorporates an entire specializer). In practice, there will be more 'specialized' methods that produce the same residual effect in connection with a particular specializer, as evidenced by the numerous binding-time improvements given in the literature. However, the universal construction may provide new insights into the nature of binding-time improvements. For instance, it will be interesting to see whether some of the non-trivial binding-time improvements (*e.g.*, [6, 2]) can be justified by the universal construction, and whether new pre-transformations can be derived from it (where the specializer incorporated in the universal construction represents the desired specialization strength). Eventually, this may lead to a better understanding of how to design and reason about binding-time improvements.

Our results do not imply that specializers which are not Jones-optimal cannot benefit from binding-time improvements. For example, it seems that the KMP test can be passed by an offline partial evaluator which is not Jones-optimal. This leads to the question what are the practical limits of specializers that are not Jones-optimal. More work will be needed to identify cause and effect.

On a more concrete level, we are not aware of an online partial evaluator that has been shown to be Jones-optimal. This question should be answered positively, as it was already done for offline partial evaluators. Also, there is not much practical experience in building Jones-optimal specializers for realistic programming languages. For instance, more work will be needed on retyping transformations for strongly typed languages to make these transformations more standard, and more program specializers truly Jones-optimal.

Acknowledgments

Comments by Sergei Abramov, Kenichi Asai, Mikhail Bulyonkov, Niels Christensen, Yukiyoshi Kameyama, Siau-Cheng Khoo, and Eijiro Sumii on an earlier version of this paper are greatly appreciated. Special thanks are due to the anonymous reviewers for thorough and careful reading of the submitted paper and for providing many valuable comments. A preliminary version of the paper was presented at the Second Asian Workshop on Programming Languages and Systems, Daejeon, South Korea.

11. 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] T. Amtoft, C. Consel, O. Danvy, K. Malmkjær. The abstraction and instantiation of string-matching programs. Technical Report BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, 2001.

[3] A. Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, Denmark, May 1993. Included in Similix distribution, 82 pages.

[4] M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 59–65. ACM Press, 1993.

[5] N. H. Christensen, R. Glück. On the equivalence of online and offline partial evaluation. Manuscript, DIKU, Department of Computer Science, University of Copenhagen, 2001.

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

[7] C. Consel, O. Danvy. For a better support of static data flow. In J. Hughes (ed.), *Functional Programming Languages and Computer Architecture. Proceedings*,

LNCS 523, 496–519. Springer-Verlag, 1991.

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

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

[10] 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.

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

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

[13] R. Glück, J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, 183–194. IEEE Computer Society Press, 1994.

[14] 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.

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

[16] J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43(2–3):149–167, 1986.

[17] 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.

[18] 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.

[19] N. D. Jones, A. Glenstrup. Program generation, termination, and binding-time analysis. In W. Taha (ed.), *Proceedings of GPCE'02*, to appear. Springer-Verlag, 2002.

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

[21] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, 258–268, Jan. 1992.

[22] 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.

[23] 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.

[24] T. Æ. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, 116–121. Yale University, 1992.

[25] 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.

[26] 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, Oct. 1985.

[27] 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.

[28] S. Skalberg. Mechanical proof of the optimality of a partial evaluator. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, 1999.

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

[30] M. Sperber, R. Glück, P. Thiemann. Bootstrapping higher-order program transformers from interpreters. In K. M. George, J. H. Carroll, D. Oppenheim, J. Hightower (eds.), *Proceedings of the 1996 ACM Symposium on Applied Computing*, 408–413. ACM Press, 1996.

[31] M. Sperber, P. Thiemann. Realistic compilation by partial evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 206–214. ACM Press, 1996.

[32] 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.

[33] P. Thiemann, R. Glück. The generation of a higher-order online partial evaluator. In M. Takeichi, T. Ida (eds.), *Fuji International Workshop on Functional and Logic Programming*, 239–253. World Scientific, 1995.

[34] P. Thiemann, M. Sperber. Polyvariant expansion and compiler generators. In D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 285–296. Springer-Verlag, 1996.

APPENDIX

A. PROPOSITIONS

This appendix gives the four propositions used in Sect. 7. They are shown by trivial constructions which are easy to verify using the definitions in Sect. 2. In the following let *bti* be a binding-time improver, *sint*, *sint*′ be self-interpreters, and *spec*, *spec*′ be specializers. Note that Jones optimality is not required for any of the propositions.

PROPOSITION 1. *For every bti there exists a pair* (*spec, sint*) *such that*

$$\forall p \in P \,.\, [\![spec]\!]\,[\,sint, SD, p\,] \;=\; [\![bti]\!]\,[\,p, SD\,]\;. \qquad (45)$$

Proof: Let *sint* be a self-interpreter, then define specializer

$$spec \stackrel{\text{def}}{=} \ulcorner \lambda(p, \mathrm{SD}, x)\,.\, \mathit{if}\ \mathit{equal}(p, \mathit{sint})\ \mathit{then}\ [\![\mathit{bti}]\!]\,[\,p, \mathrm{SD}\,] \ \mathit{else}\ [\![\mathit{spec}_{\mathit{triv}}]\!]\,[\,p, \mathrm{SD}, x\,]\urcorner\,.$$

It is easy to verify that pair $(spec, sint)$ satisfies (45). ∎

PROPOSITION 2. *For every pair* $(spec, sint)$ *there exists a* bti *such that* (45).

Proof: Define binding-time improver

$$bti \stackrel{\text{def}}{=} \ulcorner \lambda(p, \mathrm{SD})\,.\,[\![\mathit{spec}]\!]\,[\,\mathit{sint}, \mathrm{SD}, p\,]\urcorner\,.$$

It is easy to verify that bti satisfies (45). ∎

PROPOSITION 3. *For every pair* $(spec, bti)$ *there exists a pair* $(spec', sint')$ *such that*

$$\forall p \in P, \forall x \in D\,. \quad [\![\mathit{spec}']\!]\,[\,\mathit{sint}', \mathit{SSD}, p, x\,] = [\![\mathit{spec}]\!]\,[\,[\![\mathit{bti}]\!]\,[\,p, \mathit{SD}\,], \mathit{SD}, x\,]\,. \tag{46}$$

Proof: Let $sint'$ be a self-interpreter, let ${spec_{triv}}'$ be a trivial specializer for programs with two arguments such that $[\![{spec_{triv}}']\!]\,[\,p, \mathrm{SSD}, a, b\,] = \ulcorner \lambda c.[\![p]\!]\,[a, b, c]\urcorner$, then define specializer

$$spec' \stackrel{\text{def}}{=} \ulcorner \lambda(p, \mathrm{SD}, a, b)\,.\, \mathit{if}\ \mathit{equal}(p, \mathit{sint}') \ \mathit{then}\ [\![\mathit{spec}]\!]\,[\,[\![\mathit{bti}]\!]\,[\,a, \mathrm{SD}\,], \mathrm{SD}, b\,] \ \mathit{else}\ [\![{\mathit{spec}_{\mathit{triv}}}']\!]\,[\,p, \mathrm{SSD}, a, b\,]\urcorner\,.$$

It is easy to verify that pair $(spec', sint')$ satisfies (46). ∎

PROPOSITION 4. *For every pair* $(spec', sint')$ *there exists a pair* $(spec, bti)$ *such that* (46).

Proof: Let $bti = bti_{id}$ as in (5), then define specializer

$$spec \stackrel{\text{def}}{=} t\ulcorner \lambda(p, \mathrm{SD}, x)\,.\,[\![\mathit{spec}']\!]\,[\,\mathit{sint}', \mathrm{SSD}, p, x\,]\urcorner\,.$$

It is easy to verify that pair $(spec, bti)$ satisfies (46). ∎