

On Jones-Optimal Specialization for Strongly Typed Languages

Henning Makholm*

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 København Ø, Denmark
henning@makholm.net

Abstract. The phrase “optimal program specialization” was defined by Jones et al. in 1993 to capture the idea of a specializer being strong enough to remove entire layers of interpretation. As it has become clear that it does not imply “optimality” in the everyday meaning of the word, we propose to rename the concept “Jones-optimality”.

We argue that the 1993 definition of Jones-optimality is in principle impossible to fulfil for strongly typed languages due to necessary encodings on the inputs and outputs of a well-typed self-interpreter. We propose a technical correction of the definition which allows Jones-optimality to remain a meaningful concept for typed languages.

We extend recent work by Hughes and by Taha and Makholm on the long-unsolved problem of Jones-optimal specialization for strongly typed languages. The methods of Taha and Makholm are enhanced to allow “almost optimal” results when a self-interpreter is specialized to a type-incorrect program; how to do this has been an open problem since 1987. Neither Hughes’ nor Taha–Makholm’s methods are by themselves sufficient for Jones-optimal specialization when the language contains primitive operations that produce or consume complex data types. A simple postprocess is proposed to solve the problem.

An implementation of the proposed techniques has been produced and used for the first successful practical experiments with truly Jones-optimal specialization for strongly typed languages.

1 Introduction

1.1 Program Specialization

A **program specializer** is a software system that given a program and some of its input produces a new program whose behaviour on the remaining input is identical to that of the original program.

We can express this in algebraic guise:

$$\forall p, d_1, d_2 : \llbracket p \rrbracket(d_1, d_2) = \llbracket \llbracket spec \rrbracket(p, d_1) \rrbracket(d_2) \quad (1)$$

* Supported in part by the Danish National Science Research Council via project PLT (Programming Language Technology)

Here $\llbracket \cdot \rrbracket$ is the mapping that takes a program text to its meaning as a function from input to output. That is, $\llbracket p \rrbracket(d)$ denotes the output produced by the program p when it is run with input d . We call p is the **subject program** and $\llbracket spec \rrbracket(p, d_1)$ the **specialized program**.

Programs may diverge, so $\llbracket p \rrbracket(d)$ is not always defined. This means that (1) must be accompanied with a specification of how it is to be interpreted in the presence of non-termination. We think an ideal program specializer ought to have the the following properties:

- i) **Totality.** $\llbracket spec \rrbracket(p, d_1)$ should be defined for any correctly formed program p and any d_1 that can meaningfully be input to p .
- ii) **Correctness.** $\llbracket \llbracket spec \rrbracket(p, d_1) \rrbracket(d_2)$ must be defined exactly when $\llbracket p \rrbracket(d_1, d_2)$ is (in which case (1) says their values must be equal).

There is a general consensus that what we have called correctness is a good requirement. Though sometimes one sees a specializer which allows the specialized program to terminate more often than the subject program, that property is always the subject of a couple of apologising and/or defensive sentences in the accompanying paper.

The agreement about totality is much less universal. Indeed, most actual program specializers are not total; the last decade of research in automatic program specialization has shown that totality is not easily achieved together with efficiency of the specialized program and the specialization process itself. Opinions even differ about whether it is a desirable property in its own right.

Apart from the two “hard” conditions of totality and correctness there are some less formal requirements that can reasonably be applied to a program specializer:

- iii) **Strength.** The specialized program should be as least as efficient as the subject program. There should be a nontrivial class of interesting subject programs for which the specialized programs are substantially more efficient as the subject program.
- iv) **Fairness.** The behaviour of the specializer should be free of gratuitous discontinuities: Similar subject programs ought to be specialized similarly, unless there is a rational excuse for doing things differently. If the specializer can't handle some part of the subject program by its preferred technique, it should be able to fall back to a cruder method without affecting the specialization of unrelated parts of the subject program

There are many different measures of strength; in this article we focus on *Jones' optimality criterion*, as described in the next subsection.

Fairness is the most vaguely defined of the criteria we have presented. Still we maintain that it is a real and very desirable property, though we do not currently know how to characterise it completely formally.

We consider as the eventual goal of our research area to produce automatic program transformers that non-expert software developers can use with the same ease as they use other development tools such as compilers or parser generators. Thus we seek to construct specializers that are strong enough to solve real problems while still being total, correct, and fair. This does not mean that every novel solution to a subproblem *must* have all of these ideal properties initially – but we consider it a problem worth solving if a proposed technique is apparently incompatible with one of these goals.

1.2 Jones’ Optimality Criterion

Above we have assumed that all the occurring programs were written in the same fixed but arbitrary language \mathcal{L}_1 . Now let another language \mathcal{L}_2 enter the discussion and imagine that we have an \mathcal{L}_2 -interpreter *int* written in \mathcal{L}_1 . That, is, *int* should satisfy

$$\forall p \in \mathcal{L}_2\text{-programs}, \forall d : \llbracket \textit{int} \rrbracket(p, d) = \llbracket p \rrbracket_2(d) , \quad (2)$$

where $\llbracket \cdot \rrbracket_2$ is the meaning function for \mathcal{L}_2 , and each side of the equation should be defined for the same p and d .

Note in passing that (2) does not specify how *int* should behave when p is not an \mathcal{L}_2 -program. Some people maintain that a correct \mathcal{L}_2 -interpreter ought to check its input and report an error if it is not a correct program. We think this view is too restrictive; it is commonly accepted that at least the toy interpreters one usually encounters in research papers simply start interpreting their input program right away and only report errors if the program tries to actually execute a problematic statement – even if the interpreted language is supposed to be typed. We do not argue against static checking of programs, but we claim that it should be thought of as a separate activity rather than an inherent part of any interpreter.

Futamura [5] observed that one can combine an interpreter *int* with the specializer *spec* to achieve automatic translation of arbitrary programs from \mathcal{L}_2 to \mathcal{L}_1 : Combining (1) and (2) we find

$$\forall p \in \mathcal{L}_2\text{-programs}, \forall d : \llbracket \llbracket \textit{spec} \rrbracket(\textit{int}, p) \rrbracket(d) = \llbracket \textit{int} \rrbracket(p, d) = \llbracket p \rrbracket_2(d) , \quad (3)$$

that is, $\llbracket \llbracket \textit{spec} \rrbracket(\textit{int}, p) \rrbracket$ is an \mathcal{L}_1 -program that behaves identically to the \mathcal{L}_2 -program p . As before we call $\llbracket \llbracket \textit{spec} \rrbracket(\textit{int}, p) \rrbracket$ the **specialized program**, and we call the p in (3) the **source program** (cf. the “subject program” which is *int* here).

When is *spec* strong enough to make this trick interesting? A logical goal is that $\llbracket \llbracket \textit{spec} \rrbracket(\textit{int}, p) \rrbracket$ should be as efficient as if it had been manually translated from p by a human programmer who knows how the language \mathcal{L}_2 works (notice that *spec* does not know that – that knowledge is only implicit in *int* which is data for *spec*). This has been expressed by the catch phrase that the specializer “removes an entire layer of interpretation”.

Jones’ optimality criterion is a popular formalisation of the kind of strength that makes a specializer able to remove entire layers of interpretation.

The intuitive idea is that it is impossible to check that *spec* can remove *any* layer of interpretation, but we can get a good estimate of how good *spec* is at specializing away interpretation layers in general by looking at how it treats one particular interpreter, namely a self-interpreter for \mathcal{L}_1 .

A self-interpreter is an interpreter that is written in the language it interprets. Thus the effect of considering a self-interpreter is to make \mathcal{L}_2 equal \mathcal{L}_1 . Choose a self-interpreter *sint* for \mathcal{L}_1 ; Equation (3) then becomes

$$\forall p \in \mathcal{L}_1\text{-programs}, \forall d : \llbracket \llbracket \text{spec} \rrbracket (\text{sint}, p) \rrbracket (d) = \llbracket p \rrbracket (d) . \quad (4)$$

The equivalent programs p and $\llbracket \llbracket \text{spec} \rrbracket (\text{sint}, p) \rrbracket$ are both \mathcal{L}_1 -programs, so we can meaningfully compare their efficiency and thus measure how much of *sint*'s interpretative overhead has been removed by the specialization.

The original version of the optimality criterion appeared in [14, Problem 3.8] which called a specializer “strong enough” if p and $\llbracket \llbracket \text{spec} \rrbracket (\text{sint}, p) \rrbracket$ were “essentially the same program”, where “essentially” was intended to cover such minor differences as variable names and the order of function definitions, etc.

Later, Jones et al. [15, Section 6.4] refined the criterion by calling a specializer **optimal** if $\llbracket \llbracket \text{spec} \rrbracket (\text{sint}, p) \rrbracket$ is *at least as* efficient on all inputs as p . This new definition allowed the specializer to be “more than strong enough” – and indeed some of the specializers of the time were that, mostly because they sometimes continued to apply small local optimizations after having reached a specialized program that was essentially p .

This notion of “optimality” has proved to be a very productive engineering standard, but the subsequent history has also shown that it is by no means the ultimate measure of specializer strength. In particular, it has been found that even rather weak specializers can be “optimal” – for example, the Lambdamix system [9] is optimal though it does not even do value-polyvariant program-point specialization. This has caused much confusion because people (quite reasonably) think that the word “optimal” ought to imply that nothing can conceivably be better. It is too late to completely change the word, but we propose at least to warn that something uncommon is going on by saying **Jones-optimal** instead of just “optimal”.

Jones et al. [15] remark that the definition of optimality is not perfect: It can be “cheated” by a specializer which simply returns the static input if the subject program compares equal to the text of a fixed self-interpreter and otherwise specializes the subject program by a bad technique. Our reaction to this scenario is that Jones-optimality is a measure of strength alone, and it must be combined with a requirement of fairness before it really has practical value. Clearly a specializer that cheats is fundamentally unfair.

1.3 Binding-Time Analyses and Off-line Specialization

The specializers we consider in this paper are all **off-line**, meaning that the subject program presented to *spec* must be augmented with **annotations** that guide *spec* by telling it which of several methods should be employed when specializing each of the subject program's operations.

Real specializers include a **binding-time analysis** which automatically constructs suitable annotations for the subject program, so that the entire system implements a source-to-source transformation of unannotated programs.

This paper does not detail how binding-time analyses work; there is a large and still growing literature on the subject, and one of the virtues of the design we propose here is that existing binding-time analysis techniques can be reused unmodified.

1.4 Jones-Optimal Specialization for Typed Languages

This paper concerns Jones-optimal specialization for (strongly) *typed* languages. Whereas Jones-optimality was achieved quickly for untyped languages ([15] describes several examples), it was found to be a much harder problem for typed languages.

The reason for that is that a self-interpreter for a typed language is a more complex piece of software than for an untyped language. Some expressions in the interpreter must be able to represent any value the source program manipulates. These expressions must have a type; and in a strong type system there is no single type that encompasses every value a program can construct. Therefore the self-interpreter needs to represent the interpreted program’s values in an encoded form, such that the entire set of values any program can produce is injected into the set of values admitted by a single **universal type**.

Partial evaluation [15], which is the most popular technique for program specialization,¹ cannot remove the injections into (and case analyses on) the universal type from the specialized program, because partial evaluation does not in general change the type of expressions. If one expression has, for example, type integer and another has the universal type, there has to be an injection operation to connect them when the two types cannot be changed.

This problem is related, but not identical, to problems that arise when one tries to construct an (efficient) program stager – also known as a generating-extension generator, or “cogen” – for a typed language by self-applying a partial evaluator [5,15]. In fact, Jones et al. [15] present optimality chiefly as a milestone on the way to a self-applicable partial evaluator, which may have led some people to think that the *only* value of Jones-optimality was to help with self-application.

Research about program stagers for typed languages [1, 2, 16] eventually reached the conclusion that a better strategy would be to bypass self-application

¹ This terminology follows Hughes [13] but is not standard. Traditionally, there has been no need to distinguish strictly between the extensional specification of a specializer, which is (1), and its internal workings – because there has only been one basic technique: the one described by [15]. However, given that Hughes [11] as well as Danvy [4] have proposed specialization techniques that are very different from traditional ones, we think it is important to keep the “what”s separate from the “how”s. Thus we use “specialization” about anything that realizes (1) and “partial evaluation” with the implied restriction that this is done by traditional methods.

completely and write cogen by hand. We wish to point out that though this approach makes self-application unnecessary, Jones-optimality is still a desirable property in its own right, because it abstracts the ability of the specializer to produce efficient target code when one translates programs by specializing an interpreter. That is a question of strength, which is independent of whether or not the specializer uses a hand-written cogen.

1.5 Previous Solutions

In 1996 Hughes [11, 12] managed to get rid of the residual tagging and untagging operations by leaving the partial-evaluation paradigm. His **type specializer** works by constructing a type derivation for the subject program in a non-standard type system which includes information about statically known data. A specialized program is then extracted from the type derivation such that the type of each specialized expression is derived from, but not identical to, the type of an expression in the subject program.

Hughes conjectured his type specializer to be Jones-optimal after an experiment with an interpreter for a subset of the subject language showed that the interpretative overhead was removed completely. The experiment was not repeated with a complete self-interpreter, so while we find the conjecture plausible, the question has not yet been settled conclusively. In particular the termination properties of the type specializer itself are not fully understood, so it is possible that it does not always terminate when applied to a complete self-interpreter.

There is a more serious problem with the type specializer: It obtains its strength by sacrificing totality. The type specializer uses annotations on the subject program to decide precisely which tags are to be removed. If, during the specialization process, the type specializer discovers that removing these tags will harm the correctness of the specialized program, it simply gives up, outputting an error message instead of a specialized program. Hughes argues that this happens only when the source program is ill-typed (which is true) and that the lack of totality is not a problem when the goal is Jones-optimality. We think, conversely, that Jones-optimality is a measure of strength, and that strength and totality are orthogonal goals which ought to be achievable simultaneously. (Additionally, the user of the type specializer must select a trade-off between totality and strength by his selection of annotations on the subject programs; this makes it unlikely that a satisfactory, fully automatic binding-time analysis for the type specializer can be constructed).

Inspired by Hughes' work, the author and Walid Taha independently discovered that the complexity of Hughes' solution can be greatly reduced by removing the type tags as a post-process rather than during the specialization phase. That led to a joint paper [19] which proposed essentially the following specializer structure:

$$\begin{aligned}
 \text{spec}(p, d) = & \mathbf{let} \ p' = PE(p, d) \ \mathbf{in} \ \mathbf{if} \ \text{SafeToErase}(p') \\
 & \mathbf{then} \ \text{TagErasure}(p') \\
 & \mathbf{else} \ p'
 \end{aligned}$$

where PE is a traditional partial evaluator (which may include binding-time improvers, a hand-written cogen, or other existing or future refinements) that is Jones-optimal except for leaving tagging and untagging operations in the residual program p' . As with Hughes' system, the subject program p is annotated with information about which tags ought to be removed in the specialized program, but in addition the annotations try to predict which particular instance of each tag are met in each of the case analyses that are used for untagging operations. PE passes these extra annotations unchanged through to p' , after which the post-transformation *TagErasure* simply removes the indicated tagging and untagging operations (replacing untagging case analyses with the indicated branches). This tag erasure takes place only if a special type-based analysis *SafeToErase* can prove that erasing the tags will not change the behaviour of p' . If it is found that erasing tags is unsafe, p' is simply used as it is.²

The Taha–Makholm proposal sacrificed some of the strength of Hughes' system by separating the tag elimination from the specialization process, but managed to improve the totality of the specializer by using the fall-back option of not removing tags in situations where Hughes' specializer would simply reject its input. On the other hand the fact that the fall-back decision is made globally means that the design is not fair: if the tags do not match up in even a tiny bit of a big specialized program, *all* tags in the entire program will be left in.

The design does not in itself guarantee totality, because the partial evaluator PE might itself diverge – but it is *no less total* than PE . Thus unlike for Hughes' system, there is hope that a clever binding-time analysis (see, e.g., [7, 8]) might guarantee termination of PE without hurting the Jones-optimality of the entire system.

1.6 Organization of this Paper; Summary of Contributions

Section 2. We point out that necessary encodings of input and output in a typed self-interpreter prevents (4) from being true in a strict sense. We analyse this phenomenon and propose how the definition should be repaired to make Jones-optimality a meaningful property for typed languages.

Section 3. We briefly present the small PEL language, the subject language of our prototype specializer MiXIMUM which we are using for experiments in Jones-optimal specialization for typed languages.

PEL is a first-order language. The previous work [11, 12, 19] has all been formulated for higher-order languages, but we think that the restriction to a first-order language makes the essential properties of our techniques clearer. We conjecture that our techniques will be relatively easy to extend to higher-order languages.

² This description of the Taha–Makholm design is somewhat simplified. What really happens in that paper is that the tag erasure is allowed to change the behaviour of the program in a controlled way, and in the **else** branch extra “wrapper” code is added to p' to change its behaviour similarly.

Section 4. We show how the Taha–Makholm proposal can be extended to *improve the fairness* of the system while preserving Jones-optimality. The practical relevance of this is that MiXIMUM can specialize an interpreter for an untyped version of PEL with respect to a not-well-typed source program, and the specialized program will contain type tags only where absolutely necessary. MiXIMUM introduces no extraneous tags in the parts of the source program that happen to be well-typed. To the best of our knowledge this is the first serious answer to Problem 11.2 of [14]³.

Section 5. We point out a problem that has not been addressed in the previous work cited in Sect. 1.5: The previous methods work only for the cases where the embedding into the universal type can be constructed in parallel with the computation of the value that gets embedded. When the language contains primitive operations that work on compound values, the self-interpreter must contain “deep” tagging and untagging operations which cannot be eliminated totally by prior methods. We propose an “identity elimination” that can take care of this.

Section 6. MiXIMUM has been used to specialize a self-interpreter in practise, yielding Jones-optimal results for all of the source programs we tried (including the self-interpreter itself). To our knowledge, this is the first successful experiment with Jones-optimality for a typed language; the previous work has used source languages that were proper subsets of the language the interpreter was written in.

Finally, Sect. 7 concludes.

2 Jones-Optimality with a Typed Self-Interpreter

In typed languages, input and output are typed just as well as the variables inside the program. A program that expects an integer as input cannot be applied at all to a list of integers.

This has consequences for Jones-optimality experiments. Namely, it means that the self-interpreter must accept and produce the interpreted program’s input and output wrapped in the “universal type” encoding. That again means that the self-interpreter will use the same wrapping on input and output when it has been specialized – so the specialized program is not directly comparable to the source program; they do not even work on the same data!

This argument is subtle enough to deserve being recast in formulas. We use an informal notion of types to clarify what is going on. First we write down the type of a program specializer:

$$\llbracket spec \rrbracket : Pgm \times Data \rightarrow Pgm \quad , \quad (5)$$

³ The problem, reading “Achieve type security and yet do as few run time type tests as possible”, does not explicitly mention that this is to be achieved in the context of compiling by specializing an interpreter, but we consider that to be implicit in its context. Without that qualification the problem has of course long since been solved by “soft typing” systems such as [10].

where Pgm is the type of all program texts and $Data$ is whatever type $spec$ uses for encoding the static input to a program (assume that $spec$ itself is written in a typed language and so need an encoding here).

Equation (5) is correct but does not tell the whole story. In particular, the second argument to $spec$ must encode a value of the type the program given as the first argument to $spec$ expects as its first argument. And there is a certain relation between the types of the subject program and the specialized program.

To express this, we introduce a notion of subtyping – reminiscent of the “underbar types” used by [15, Section 16.2] – into our informal type language: $\frac{\alpha \rightarrow \beta}{Pgm}$ is the type of the *text* of a program which it itself has type $\alpha \rightarrow \beta$.

That is, $\frac{\alpha \rightarrow \beta}{Pgm}$ is a subtype of Pgm . Similarly, let $\frac{\alpha}{Data}$ be the subtype of $Data$ consisting of the encodings of all values of type α .

With this notation we can refine (5) to

$$\forall \alpha, \beta, \gamma : \llbracket spec \rrbracket : \frac{\alpha \times \beta \rightarrow \gamma}{Pgm} \times \frac{\alpha}{Data} \rightarrow \frac{\beta \rightarrow \gamma}{Pgm} . \quad (6)$$

Notice that if we remove everything above the bars we arrive back at (5).

Similarly to (6) we can give the type of the self-interpreter:

$$\forall \alpha, \beta : \llbracket sint \rrbracket : \frac{\alpha \rightarrow \beta}{Pgm} \times \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ} \quad (7)$$

This is the type of the *meaning* of the self-interpreter. The type of the *text* of the self-interpreter (which we give as input to the specializer) has a third layer:

$$\forall \alpha, \beta : \llbracket sint \rrbracket : \frac{\frac{\alpha \rightarrow \beta}{Pgm} \times \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ}}{Pgm} \quad (8)$$

Now say that we want to compute $\llbracket spec \rrbracket(sint, d)$ for some d . Equations (6) and (8) imply that d must have type $\frac{\alpha \rightarrow \beta}{Data}$; that is, d must be the *Data* encoding of a program p . However $\llbracket p \rrbracket$ has type $\alpha \rightarrow \beta$ whereas

$$\llbracket \llbracket spec \rrbracket(sint, d) \rrbracket : \frac{\alpha}{Univ} \rightarrow \frac{\beta}{Univ} , \quad (9)$$

so the comparison between p and $\llbracket spec \rrbracket(sint, d)$ that is the core of Jones’ optimality criterion is inherently unfair. For some programs p , for example one that efficiently constructs a list of the first n natural numbers, $\llbracket spec \rrbracket(sint, d)$ is *necessarily* slower than p , simply because it has to construct a bigger output value. Thus Jones-optimal specialization of an encoding interpreter is impossible!

Earlier work does not, as far as we are aware, address this problem directly. Nevertheless, it has been dealt with in different ways.

Hughes allows his type specializer to replace the types of the specialized program’s input and output with *specialized* versions of the subject program’s

types. This means that the extensional specification of the specializer (1) is not quite true; $\llbracket \llbracket spec \rrbracket(p, d_1) \rrbracket(d_2)$ must be replaced by $W(\llbracket \llbracket spec \rrbracket(p, d_1) \rrbracket(U(d_2))$), where U and W are coercions that can be derived from a “residual type” that is a by-product of the type specialization. It may be argued that when we translate programs by specializing interpreters we *want* the type to change to reflect the type of the source program. However from our perspective the eventual goal is a generally usable specializer which simply happens to be strong enough to specialize interpreters well. Allowing a general specializer to make its own guess at which type we want for the specialized program would make it difficult to automate the use of the specialized program.

The earlier Taha–Makholm design [19] modified this solution by suggesting that the specification of the U and W coercions should be an *input* to the specialization process rather than something the specializer decides for itself. That is better, because it gives control back to the user. But we still consider it clumsy to expect the user of a generally usable specializer to learn a separate notation for specifying how the type of the specialized program should relate to the subject program’s type.

So instead of changing the behaviour of the specializer, we propose to solve the problem by using a different interpreter in the optimality test. Instead of a self-interpreter with type (7) we use a variant $sint_{\alpha \rightarrow \beta}$ with type

$$\llbracket sint_{\alpha \rightarrow \beta} \rrbracket : \frac{\alpha \rightarrow \beta}{Pgm} \times \alpha \rightarrow \beta . \quad (10)$$

The interpreter text $sint_{\alpha \rightarrow \beta}$ must depend on α and β , but once we know these it is easy to construct $sint_{\alpha \rightarrow \beta}$ mechanically:

$$sint_{\alpha \rightarrow \beta} = decode_{\beta} \circ sint \circ \langle Id, encode_{\alpha} \rangle , \quad (11)$$

where $encode_{\alpha} : \alpha \rightarrow Univ$ and $decode_{\beta} : Univ \rightarrow \beta$ are the natural injection and extraction functions, and “ \circ ” is symbolic function composition – implemented naïvely by simply generating a program whose main function applies the two composants to the input sequentially.

We propose that this construction be used to adjust Jones’ optimality criterion when used in a typed context: a specializer for a typed language should be called **Jones-optimal** if for any types α, β and any program $p : \frac{\alpha \rightarrow \beta}{Pgm}$, the program $\llbracket spec \rrbracket(sint_{\alpha \rightarrow \beta}, p)$ is at least as efficient as p on any input.

This solution is more general than the one proposed by [19], because it can be seen as simply programming the U and W coercions into the subject program before the specialization. Our solution means that the user of the specializer does not have to learn an independent notation for specifying the coercions. It is also more flexible because the subject language is generally more expressive than a special-purpose notation for coercions. On the other hand it requires more from the specializer to obtain Jones-optimality with this strategy – but in Sect. 5 we show that the required strength is in general necessary for other reasons anyway.

$p ::= f x = e; p?$	Program
$e ::= \bullet$	Unit value constant
c	Integer constant
$(e \diamond e)$	Integer operation
(e, e)	Pair construction
$\text{fst } e$	Pair destruction
$\text{snd } e$	Pair destruction
$L e$	Sum construction
$R e$	Sum construction
$\text{case } e \text{ of } \left\{ \begin{array}{l} L x \mapsto e \\ R x \mapsto e \end{array} \right\}$	Sum destruction
x	Variable reference
error	Error indication
$f e$	Function application
$\text{let } x = e \text{ in } e \text{ end}$	Let-binding
$\diamond ::= + \mid - \mid * \mid =$	Integer operators
$f \in \langle \text{Func} \rangle$	Function names
$x \in \langle \text{Var} \rangle$	Variable names
$c \in \mathbb{N} = \{0, 1, 2, \dots\}$	The integers

Fig. 1. Abstract syntax of PEL

3 Language

We use the PEL language defined by Welinder [20], extended for technical reasons with an explicit unit type. It is a minimal, call-by-value functional language which manipulates first-order values built from this grammar:

$$\langle \text{Val} \rangle \ni v ::= \bullet \mid 0 \mid 1 \mid 2 \mid \dots \mid (v, v) \mid L v \mid R v$$

The language is attractive for our purposes because of its simplicity and because we can use Welinder’s PEL self-interpreter, which has been formally proven correct, in optimality experiments. Using an existing self-interpreter⁴ should vindicate us of the possible critique of having programmed the self-interpreter in unnatural ways to compensate for idiosyncrasies in the specializer.

The syntax of PEL expressions is given in Fig. 1, and the dynamic semantics in Fig. 2. A program is a set of mutually recursive functions, the first of which defines the meaning of the program – the only nontrivial feature of the semantics is that the equality test operator “=” delivers its result encoded using the sum type; $L \bullet$ represents false and $R \bullet$ represents true. Thus the case expression can be used as an if-then-else construct. For full details see [20, Section 3.3].

⁴ The self-interpreter in [20] does not use or support the “ \bullet ” value; Welinder kindly added support for \bullet to his interpreter and adjusted his mechanical proof of its correctness to work with the new interpreter.

$$\begin{array}{c}
\frac{}{E \vdash_p \bullet \Rightarrow \bullet} \qquad \frac{}{E \vdash_p c \Rightarrow c} \\
\frac{E \vdash_p e_1 \Rightarrow c_1 \quad E \vdash_p e_2 \Rightarrow c_2}{E \vdash_p (e_1 \diamond e_2) \Rightarrow v} \quad v = c_1 \diamond c_2 \qquad \frac{E \vdash_p e_1 \Rightarrow v_1 \quad E \vdash_p e_2 \Rightarrow v_2}{E \vdash_p (e_1, e_2) \Rightarrow (v_1, v_2)} \\
\frac{E \vdash_p e \Rightarrow (v_1, v_2)}{E \vdash_p \text{fst } e \Rightarrow v_1} \qquad \frac{E \vdash_p e \Rightarrow (v_1, v_2)}{E \vdash_p \text{snd } e \Rightarrow v_2} \\
\frac{E \vdash_p e \Rightarrow v}{E \vdash_p L e \Rightarrow L v} \qquad \frac{E \vdash_p e_0 \Rightarrow L v_1 \quad E\{x_1 \mapsto v_1\} \vdash_p e_1 \Rightarrow v'}{E \vdash_p \text{case } e_0 \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \Rightarrow v'} \\
\frac{E \vdash_p e \Rightarrow v}{E \vdash_p R e \Rightarrow R v} \qquad \frac{E \vdash_p e_0 \Rightarrow R v_2 \quad E\{x_2 \mapsto v_2\} \vdash_p e_2 \Rightarrow v'}{E \vdash_p \text{case } e_0 \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \Rightarrow v'} \\
\frac{}{E \vdash_p x \Rightarrow E(x)} \qquad \frac{E \vdash_p e_0 \Rightarrow v \quad \{x \mapsto v\} \vdash_p e_1 \Rightarrow v' \quad [f \ x = e_1] \in p}{E \vdash_p f e_0 \Rightarrow v'} \\
\frac{E \vdash_p e_0 \Rightarrow v \quad E\{x \mapsto v\} \vdash_p e_1 \Rightarrow v'}{E \vdash_p \text{let } x = e_0 \text{ in } e_1 \text{ end} \Rightarrow v'}
\end{array}$$

Fig. 2. Dynamic semantics for PEL. There is intentionally no rule for error

3.1 Type System

Because we are concerned with specialization of typed languages, we add a (monomorphic) type system to PEL. A **type** in our system is a finite or (regularly) infinite tree built from this grammar:

$$\langle Typ \rangle \ni \tau ::= \text{unit} \mid \text{int} \mid \langle L \tau + R \tau \rangle \mid (\tau, \tau)$$

Our implementation represents infinite types as graphs; this representation is isomorphic to Welinder’s construction (which uses explicit syntax for recursion and a co-inductively defined equivalence relation) but more convenient in implementations.

The typing rules for PEL expressions are given in Fig. 3. They define a judgement “ $\Gamma \vdash_T e : \tau$ ” where $\Gamma : \langle Var \rangle \rightarrow \langle Typ \rangle$ and $T : \langle Func \rangle \rightarrow \langle Typ \rangle \times \langle Typ \rangle$ are type environments that give types to the program’s variables and functions, respectively.

4 An Erasure Analysis for Improving Fairness

Recall the specializer structure proposed by [19]:

$$\begin{aligned}
\text{spec}(p, d) = \text{let } p' = PE(p, d) \text{ in if } \text{SafeToErase}(p') \\
\text{then } \text{TagErasure}(p') \\
\text{else } p'
\end{aligned}$$

$\overline{\Gamma \vdash_{\mathcal{T}} \bullet : \text{unit}}$	$\overline{\Gamma \vdash_{\mathcal{T}} c : \text{int}}$	
$\frac{\Gamma \vdash_{\mathcal{T}} e_1 : \text{int} \quad \Gamma \vdash_{\mathcal{T}} e_2 : \text{int}}{\Gamma \vdash_{\mathcal{T}} (e_1 \diamond e_2) : \text{int}} \diamond \neq =$	$\frac{\Gamma \vdash_{\mathcal{T}} e_1 : \text{int} \quad \Gamma \vdash_{\mathcal{T}} e_2 : \text{int}}{\Gamma \vdash_{\mathcal{T}} (e_1 = e_2) : \langle \text{L unit} + \text{R unit} \rangle}$	
$\frac{\Gamma \vdash_{\mathcal{T}} e_1 : \tau_1 \quad \Gamma \vdash_{\mathcal{T}} e_2 : \tau_2}{\Gamma \vdash_{\mathcal{T}} (e_1, e_2) : (\tau_1, \tau_2)}$	$\frac{\Gamma \vdash_{\mathcal{T}} e : (\tau_1, \tau_2)}{\Gamma \vdash_{\mathcal{T}} \text{fst } e : \tau_1}$	$\frac{\Gamma \vdash_{\mathcal{T}} e : (\tau_1, \tau_2)}{\Gamma \vdash_{\mathcal{T}} \text{snd } e : \tau_2}$
$\frac{\Gamma \vdash_{\mathcal{T}} e : \tau_1}{\Gamma \vdash_{\mathcal{T}} \text{L } e : \langle \text{L } \tau_1 + \text{R } \tau_2 \rangle}$	$\frac{\Gamma \vdash_{\mathcal{T}} e : \tau_2}{\Gamma \vdash_{\mathcal{T}} \text{R } e : \langle \text{L } \tau_1 + \text{R } \tau_2 \rangle}$	
$\frac{\Gamma \vdash_{\mathcal{T}} e_0 : \langle \text{L } \tau_1 + \text{R } \tau_2 \rangle \quad \Gamma \{x_1 \mapsto \tau_1\} \vdash_{\mathcal{T}} e_1 : \tau \quad \Gamma \{x_2 \mapsto \tau_2\} \vdash_{\mathcal{T}} e_2 : \tau}{\Gamma \vdash_{\mathcal{T}} \text{case } e_0 \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto e_1 \\ \text{R } x_2 \mapsto e_2 \end{array} \right\} : \tau}$		
$\overline{\Gamma \vdash_{\mathcal{T}} x : \Gamma(x)}$	$\overline{\Gamma \vdash_{\mathcal{T}} \text{error} : \tau}$	
$\frac{\Gamma \vdash_{\mathcal{T}} e_0 : \tau_0}{\Gamma \vdash_{\mathcal{T}} f e_0 : \tau_1} T(f) = \tau_0 \rightarrow \tau_1$	$\frac{\Gamma \vdash_{\mathcal{T}} e_0 : \tau_0 \quad \Gamma \{x \mapsto \tau_0\} \vdash_{\mathcal{T}} e_1 : \tau_1}{\Gamma \vdash_{\mathcal{T}} \text{let } x = e_0 \text{ in } e_1 \text{ end} : \tau_1}$	

Fig. 3. Typing rules for PEL

The global test on $\text{SafeToErase}(p')$ hurts the fairness of the system, so in MiX-IMUM we use a different structure

$$\begin{aligned} \text{spec}(p, d) = & \text{let } p' = PE(p, d) \\ & p'' = \text{ErasureAnalysis}(p') \\ & \text{in } \text{TagErasure}(p'') \end{aligned}$$

The annotations on p are now only used to guide the partial evaluator, and p' is an unannotated program. The analysis ErasureAnalysis then analyses p' to find out which injections and case analyses can safely be erased without changing the meaning of p' . The output of ErasureAnalysis is an annotated program p'' which except for the annotations that guide TagErasure is the same program as p' .

The advantage of this design is that it allows the erasure annotations to be more fine-grained than in the Taha–Makholm proposal. There the erasure annotations on p' came from the annotated subject program p , so two tagging operations in p' which originated in the same tagging operation in p had to have the same annotation. MiXIMUM can give them different erasure annotations because the erasure annotations are created only after the partial evaluation.

The erasure analysis works by doing a non-standard type inference using the non-standard type system defined in Fig. 4. Here types (finite or infinite) are built from the grammar

$$\begin{aligned} \langle \text{STyp} \rangle \ni \tilde{\tau} &::= \text{unit} \mid \text{int} \mid \langle \text{L}_\sigma \tilde{\tau} + \text{R}_\sigma \tilde{\tau} \rangle \mid (\tilde{\tau}, \tilde{\tau}) \\ \langle \text{Sign} \rangle \ni \sigma &::= \oplus \mid \ominus \end{aligned}$$

$$\begin{array}{c}
\overline{\check{I} \vdash_{\check{T}} \bullet : \text{unit}} \qquad \overline{\check{I} \vdash_{\check{T}} c : \text{int}} \\
\frac{\check{I} \vdash_{\check{T}} e_1 : \text{int} \quad \check{I} \vdash_{\check{T}} e_2 : \text{int}}{\check{I} \vdash_{\check{T}} (e_1 \diamond e_2) : \text{int}} \diamond \neq = \frac{\check{I} \vdash_{\check{T}} e_1 : \text{int} \quad \check{I} \vdash_{\check{T}} e_2 : \text{int}}{\check{I} \vdash_{\check{T}} (e_1 = e_2) : \langle L_{\oplus} \text{unit} + R_{\oplus} \text{unit} \rangle} \\
\frac{\check{I} \vdash_{\check{T}} e_1 : \check{\tau}_1 \quad \check{I} \vdash_{\check{T}} e_2 : \check{\tau}_2}{\check{I} \vdash_{\check{T}} (e_1, e_2) : (\check{\tau}_1, \check{\tau}_2)} \quad \frac{\check{I} \vdash_{\check{T}} e : (\check{\tau}_1, \check{\tau}_2)}{\check{I} \vdash_{\check{T}} \text{fst } e : \check{\tau}_1} \quad \frac{\check{I} \vdash_{\check{T}} e : (\check{\tau}_1, \check{\tau}_2)}{\check{I} \vdash_{\check{T}} \text{snd } e : \check{\tau}_2} \\
\frac{\check{I} \vdash_{\check{T}} e : \check{\tau}_1}{\check{I} \vdash_{\check{T}} L e : \langle L_{\oplus} \check{\tau}_1 + R_{\sigma} \check{\tau}_2 \rangle} \quad \frac{\check{I} \vdash_{\check{T}} e : \check{\tau}_2}{\check{I} \vdash_{\check{T}} R e : \langle L_{\sigma} \check{\tau}_1 + R_{\oplus} \check{\tau}_2 \rangle} \\
\frac{\check{I} \vdash_{\check{T}} e_0 : \langle L_{\sigma_1} \check{\tau}_1 + R_{\sigma_2} \check{\tau}_2 \rangle \quad \check{I} \{x_1 \mapsto \check{\tau}_1\} \vdash_{\check{T}} e_1 : \check{\tau} \quad \check{I} \{x_2 \mapsto \check{\tau}_2\} \vdash_{\check{T}} e_2 : \check{\tau}}{\check{I} \vdash_{\check{T}} \text{case } e_0 \text{ of } \left\{ \begin{array}{l} L_{x_1 \mapsto e_1} \\ R_{x_2 \mapsto e_2} \end{array} \right\} : \check{\tau}} \\
\overline{\check{I} \vdash_{\check{T}} x : \check{I}(x)} \quad \overline{\check{I} \vdash_{\check{T}} \text{error} : \check{\tau}} \\
\frac{\check{I} \vdash_{\check{T}} e_0 : \check{\tau}_0}{\check{I} \vdash_{\check{T}} f e_0 : \check{\tau}_1} \check{I}(f) = \check{\tau}_0 \rightarrow \check{\tau}_1 \quad \frac{\check{I} \vdash_{\check{T}} e_0 : \check{\tau}_0 \quad \check{I} \{x \mapsto \check{\tau}_0\} \vdash_{\check{T}} e_1 : \check{\tau}_1}{\check{I} \vdash_{\check{T}} \text{let } x = e_0 \text{ in } e_1 \text{ end} : \check{\tau}_1}
\end{array}$$

Fig. 4. Non-standard type system for the erasure analysis. Apart from the signs in the sum types these rules are identical to the ones in Fig. 3

The intuitive meaning of the σ s in the sum types is that, for example, the type $\langle L_{\oplus} \check{\tau}_1 + R_{\ominus} \check{\tau}_2 \rangle$ is a sum type where only the $L \check{\tau}_1$ component is ever constructed. Such a type can safely be replaced with $\check{\tau}_1$ and the injections and case analyses for that sum type can be removed.

A non-standard type derivation for the program can be constructed by a straightforward unification-based type inference (which uses a graph unification algorithm to construct recursive types where necessary). The unifications never fail: We assume the program to be conventionally well-typed before the analysis, so there is at least one valid derivation by the rules in Fig. 4, given by attaching \oplus s to each sum type in the conventional derivation (note that none of the typing rules require any σ to be \ominus).

In general, however, some σ variables have not yet been unified with \oplus after the non-standard type inference. These are then artificially set to \ominus (except that all σ s that appear in the type of the main function must be \oplus , which means that no tags that can be seen from outside the program will be erased). The result is a “maximally \oplus -free” typing for the program.

The actual tag erasure now consists of applying the $\llbracket \cdot \rrbracket$ mapping defined in Fig. 5 to each function body in the program. This mapping is guided by the type annotations from the non-standard type analysis.

We remark that the work necessary to do the erasure analysis we have presented does not significantly exceed the work needed to compute the *SafeToErase*

$$\begin{aligned}
\llbracket \bullet \rrbracket &= \bullet \\
\llbracket c \rrbracket &= c \\
\llbracket (e_1 \diamond e_2) \rrbracket &= (\llbracket e_1 \rrbracket \diamond \llbracket e_2 \rrbracket) \\
\llbracket (e_1, e_2) \rrbracket &= (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\llbracket \text{fst } e \rrbracket &= \text{fst } \llbracket e \rrbracket \\
\llbracket \text{snd } e \rrbracket &= \text{snd } \llbracket e \rrbracket \\
\llbracket (L e)^{\oplus\oplus} \rrbracket &= L \llbracket e \rrbracket \\
\llbracket (L e)^{\oplus\ominus} \rrbracket &= \llbracket e \rrbracket \\
\llbracket (R e)^{\oplus\oplus} \rrbracket &= R \llbracket e \rrbracket \\
\llbracket (R e)^{\oplus\ominus} \rrbracket &= \llbracket e \rrbracket \\
\llbracket \text{case } e_0^{\oplus\oplus} \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \rrbracket &= \text{case } \llbracket e_0 \rrbracket \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto \llbracket e_1 \rrbracket \\ R x_2 \mapsto \llbracket e_2 \rrbracket \end{array} \right\} \\
\llbracket \text{case } e_0^{\oplus\ominus} \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \rrbracket &= \text{let } x_1 = \llbracket e_0 \rrbracket \text{ in } \llbracket e_1 \rrbracket \text{ end} \\
\llbracket \text{case } e_0^{\ominus\oplus} \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \rrbracket &= \text{let } x_2 = \llbracket e_0 \rrbracket \text{ in } \llbracket e_2 \rrbracket \text{ end} \\
\llbracket \text{case } e_0^{\ominus\ominus} \text{ of } \left\{ \begin{array}{l} L x_1 \mapsto e_1 \\ R x_2 \mapsto e_2 \end{array} \right\} \rrbracket &= \text{let } x = \llbracket e_0 \rrbracket \text{ in error end} \\
\llbracket x \rrbracket &= x \\
\llbracket \text{error} \rrbracket &= \text{error} \\
\llbracket f e \rrbracket &= f \llbracket e \rrbracket \\
\llbracket \text{let } x = e_0 \text{ in } e_1 \text{ end} \rrbracket &= \text{let } x = \llbracket e_0 \rrbracket \text{ in } \llbracket e_1 \rrbracket \text{ end}
\end{aligned}$$

Fig. 5. Specification of the tag erasure. The expressions inside the $\llbracket \cdot \rrbracket$ s are supposed to be annotated with sum-reduction types; the pattern “ $e^{\sigma_1 \sigma_2}$ ” matches an e that annotated with type $\langle L_{\sigma_1} \tilde{\tau}_1 + R_{\sigma_2} \tilde{\tau}_2 \rangle$ for some $\tilde{\tau}_{1,2}$

function in the Taha–Makholm proposal; essentially our new design improves fairness for free!

5 Identity Elimination

The design presented in the previous section is not actually strong enough to specialize PEL Jones-optimally. The problem is PEL’s “=” operator which takes two int operands and produces an $\langle L \text{ unit} + R \text{ unit} \rangle$ result. The self-interpreter must inject this compound result into the universal type, which means that the source expression $(e_1 = e_2)$ after the initial partial evaluation becomes

$$\text{case } (e_1 = e_2) \text{ of } \left\{ \begin{array}{l} L x \mapsto R R R L L \bullet \\ R x \mapsto R R R L \bullet \end{array} \right\} \quad (12)$$

which can be broken down as

$$\text{case } (e_1 = e_2) \text{ of } \left\{ \begin{array}{l} L x \mapsto \text{TagAsSum}(L \text{ TagAsUnit}(\bullet)) \\ R x \mapsto \text{TagAsSum}(R \text{ TagAsUnit}(\bullet)) \end{array} \right\} \quad (13)$$

or just

$$\text{encode}_{(\text{L unit}+\text{R unit})}(e_1 = e_2) . \quad (14)$$

After tag elimination that becomes

$$\text{case } (e_1 = e_2) \text{ of } \left\{ \begin{array}{l} \text{L } x \mapsto \text{L } \bullet \\ \text{R } x \mapsto \text{R } \bullet \end{array} \right\} , \quad (15)$$

where the (by now superfluous) case analysis makes the expression less efficient than the source expression $(e_1 = e_2)$. Therefore the result is not Jones-optimal.

Note that the resulting expression (15) is independent of the particular value encoding used by the self-interpreter. It may look like the “encoding of booleans as tagged units” comes from the self-interpreter, but the fact is that this “encoding” is part of the language specification – and the value encoding that the self-interpreter puts *atop* that one disappeared in the tag elimination phase that took (12) to (15).

The moral of this example is that though tag elimination in itself can remove tagging and untagging operations, it cannot completely remove entire encode_τ or decode_τ computations. They get reduced to code that meticulously breaks a value apart to its primitive components, only to build an identical value from scratch!

It is seen that a situation similar to (15) will arise in natural self-interpreters for *any* primitive operation that delivers or expects a value of a complex type. In PEL the only occurrence of the problem is with the equality test, but other languages might have primitive operations for arbitrary complex types; each must be accompanied by an encode_τ or decode_τ in a self-interpreter. In general these conversions may even be recursive: Consider for example a primitive operation that finds the least element of a list of integers. Then the self-interpreter’s case for that function must use a recursive function $\text{decode}_{[\text{int}]}$ which after tag erasure becomes a recursive identity mapping which might look like

$$f_{\text{id}} x = \text{case } x \text{ of } \left\{ \begin{array}{l} \text{L } y \mapsto \text{L } \bullet \\ \text{R } z \mapsto \text{R } (\text{fst } z, f_{\text{id}} \text{snd } z) \end{array} \right\} . \quad (16)$$

Our solution is yet another postprocess which we call **identity elimination**. This phase solves not only the problem outlined above, but also takes care of the encode_α and decode_β components of (11), which the tag elimination also reduce to identity mappings akin to (16).

The identity elimination proceeds in several phases. In the first phase, every \bullet in the program is replaced with the innermost enclosing variable whose type is unit, if such one exists. This transforms (16) to

$$f_{\text{id}} x = \text{case } x \text{ of } \left\{ \begin{array}{l} \text{L } y \mapsto \text{L } y \\ \text{R } z \mapsto \text{R } (\text{fst } z, f_{\text{id}} \text{snd } z) \end{array} \right\} \quad (17)$$

which makes it more explicit that f_{id} is really a identity function. The f_{id} defined by (16) is not an identity function when applied to, say, $\text{R}(4, \text{L } 2)$ – but replacing the \bullet by y localizes the global knowledge that f_{id} will not be applied to such a

$$\begin{array}{l}
(\text{fst } \pi_1 \dots \pi_n x, \text{snd } \pi_1 \dots \pi_n x) \rightarrow \pi_1 \dots \pi_n x \quad \text{where each } \pi_i \in \{\text{fst}, \text{snd}\} \\
\text{case } e \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto \text{L } x_1 \\ \text{R } x_2 \mapsto \text{R } x_2 \end{array} \right\} \rightarrow e \\
\text{case } e \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto \text{L } e_1 \\ \text{R } x_2 \mapsto \text{L } e_2 \end{array} \right\} \rightarrow \text{L case } e \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto e_1 \\ \text{R } x_2 \mapsto e_2 \end{array} \right\} \\
\text{case } e \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto \text{R } e_1 \\ \text{R } x_2 \mapsto \text{R } e_2 \end{array} \right\} \rightarrow \text{R case } e \text{ of } \left\{ \begin{array}{l} \text{L } x_1 \mapsto e_1 \\ \text{R } x_2 \mapsto e_2 \end{array} \right\} \\
f e \rightarrow e \quad \text{if } f \in \mathcal{F}
\end{array}$$

Fig. 6. Reduction rules for the identity elimination. The two first rules might be construed a form of η -reduction for product and sum types. In the last rule \mathcal{F} is a set of names of functions which are supposed to be identity functions

value. This phase could be avoided by consistently writing all *encode* and *decode* operations to reuse unit values, but that would hardly be a natural programming style.

The main part of the identity elimination consists of repeatedly applying the reductions in Fig. 6 to the program. The last of these reductions refers to a set \mathcal{F} of names of functions that are supposed to be identity functions. Intuitively, the functions meant to be in \mathcal{F} are the ones like f_{id} that are used for the necessary recursion in *encode* $_{\tau}$ or *decode* $_{\tau}$ when τ is recursive.

Formally we want \mathcal{F} to be a safe approximation to the set of functions f such that $f e$ and e are always equivalent (assuming that $f e$ is well-typed). How can we compute such an \mathcal{F} ? A first attempt would be to use the largest \mathcal{F} such that \mathcal{F} allows the definitions for all $f \in \mathcal{F}$ to be reduced to “ $f x = x$ ” – but that alone would not be sound, because it would allow the function

$$f_0 x = f_0 x \tag{18}$$

to be classified as an identity function, despite being non-terminating for all inputs.

We do not have an *elegant* way to compute \mathcal{F} , but we do have a safe one that seems to work in practise: \mathcal{F} must allow the definitions for all $f \in \mathcal{F}$ to be reduced to “ $f x = x$ ” under the constraint that the reduction $f e \rightarrow e$ may be used only if $f \in \mathcal{F}$ and simple syntactic conditions can ensure that the value of e is always smaller than the argument to the function where the reduction is applied. For example, in (17) that is true of the call to f_{id} because the argument is $\text{fst } z$, which is smaller than z , which by the case expression is smaller than x .

With this condition a largest possible \mathcal{F} can be found by fixed-point iteration. In the final reduction after a safe \mathcal{F} has been computed, we allow $f e \rightarrow e$ to be applied whenever $f \in \mathcal{F}$.

Finally, any remaining references to variables of type *unit* are replaced with \bullet expressions. Aside from making the program more readable, this means that the entire identity elimination never makes a program *less* efficient under the not

unreasonable assumption that constructing a constant • might be faster than retrieving it from a variable.

6 Experiments with MiXIMUM

MiXIMUM is a prototype implementation of the design presented above. The system is implemented in Moscow ML [18] and is available electronically as <http://www.diku.dk/~makholm/miximum.tar.gz>⁵.

We have used MiXIMUM to perform the Jones-optimality experiment described in Sect. 2 with Welinder’s self-interpreter [20] and a number of source programs p , the biggest of which was the self-interpreter itself. In each case the result of specializing the self-interpreter to p was alpha-equivalent to the result of running p itself through MiXIMUM’s post-processes, none of which can degrade the performance of the program. In other words, MiXIMUM meets the definition of Jones-optimality – at least in the examples we have tried.

MiXIMUM takes care to preserve the subject program’s termination properties, that is, to be “correct” in the sense of Sect. 1.1. The partial evaluator never discards code or reduces let bindings, so a subject expression such as

```
let x = 17 in (x + 42) end
```

will, as a first approximation, be specialized to

```
let x = 17 in let x' = x in let x'' = 42 in 59 end end end
```

and after the main specialization an eventual reduction pass reduces all let bindings where it can be proved (by simple syntactic conditions) that reduction can neither change the termination properties of the program nor degrade its efficiency. That reduces the example expression above to the constant expression 59.

It is hardly surprising that such simple techniques preserve the subject program’s termination properties, but we can offer the less trivial observation that they apparently do not interfere with Jones-optimality. In particular, the let bindings that result from erasing an untagging operation (Fig. 5) appear always to be eligible for reduction.

⁵ Readers who download MiXIMUM will discover that the system is not quite faithful to the design in Sect. 4. The partial evaluator in the prototype has been split into two phases, the second of which occurs after the tag erasure. This has historical reasons, and we’re presently working on bringing the structure of MiXIMUM closer to the ideal structure we have presented here. There is also another postprocess which unfolds spurious let bindings and functions as a local arity raiser. This was found to be necessary for achieving Jones-optimality but does not in itself contain novel ideas.

7 Conclusion

We have shown how to extend the design of Taha and Makhholm [19] to be more fair without sacrificing Jones-optimality. We have implemented our proposed design and conducted experiments which strongly suggest that it is Jones-optimal.

Two directions of further work immediately suggest themselves here: to extend the MiXIMUM implementation to work with a higher-order language, and to produce actual proofs of its correctness and Jones-optimality. Both of these lines of work are being investigated presently.

A third direction would be to investigate how our methods could be extended to deal with languages with user-defined n -ary sum types. The simplicity of the methods presented here are to some extent due to the fact that PEL has only binary sums. Achieving Jones-optimality with a self-interpreter that handles arbitrary sum types would probably require adding a variant of constructor specialization [17] and integrate it with the methods presented here. (We note that Hughes' type specializer [11] does include constructor specialization).

Acknowledgements

I thank Arne Glenstrup, John Hughes, Neil D. Jones, Walid Taha, Peter Thiemann, and Morten Welinder for constructive discussions and helpful advice on presenting and improving the results reported here. Thanks also to the anonymous referees who pointed out numerous points in the draft version that needed better explanation.

References

1. Andersen, L. O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, Department of Computer Science, University of Copenhagen (1994). DIKU-TR-94/19
2. Birkedal, L. and Welinder, M.: Hand-writing program generator generators. In: Hermenegildo, M. and Penjam, J. (eds.): Programming Language Implementation and Logic Programming: 6th International Symposium, PLILP '94. Lecture Notes in Computer Science, Vol. 844. Springer-Verlag, Berlin Heidelberg New York, 198–214. <ftp://ftp.diku.dk/diku/semantics/papers/D-199.dvi.Z>
3. Bjørner, D., Ershov, A. P., and Jones, N. D. (eds.): Partial Evaluation and Mixed Computation: selected papers from the IFIP TC2 Workshop, October 1987. Special issue of *New Generation Computing* **6:2–3**
4. Danvy, O.: Type-directed partial evaluation. In: Principles of Programming Languages: 23rd ACM SIGPLAN-SIGACT Symposium, POPL '96. ACM Press, New York, NY, USA, 242–257. <ftp://ftp.daimi.au.dk/pub/empl/danvy/Papers/danvy-popl96.ps.gz>
5. Futamura, Y.: Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls* **2** (1971) 45–50. Reprinted as [6]
6. Futamura, Y.: Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation* **14** (1999) 381–391. Reprint of [5]

7. Glenstrup, A. J.: Terminator II: Stopping Partial Evaluation of Fully Recursive Programs. Master's thesis, Department of Computer Science, University of Copenhagen (1999). DIKU-TR-99/8. (<http://www.diku.dk/~panic/TerminatorII/>)
8. Glenstrup, A. J. and Jones, N. D.: BTA algorithms to ensure termination of off-line partial evaluation. In: Bjørner, D., Broy, M., and Pottosin, I. V. (eds.): Perspectives of System Informatics: 2nd International Andrei Ershov Memorial Conference, PSI '96. Lecture Notes in Computer Science, Vol. 1181. Springer-Verlag, Berlin Heidelberg New York, 273–284.
(<ftp://ftp.diku.dk/diku/semantics/papers/D-274.ps.gz>)
9. Gomard, C. K. and Jones, N. D.: A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming* **1** (1991) 21–69
10. Henglein, F.: Global tagging optimization by type inference. In: *Lisp and Functional Programming: 1992 ACM Conference, LFP '92*. Special issue of *ACM LISP Pointers* **5:1**. ACM Press, New York, NY, USA, 205–215
11. Hughes, J.: Type specialization for the λ -calculus; or, A new paradigm for partial evaluation based on type inference. In: Danvy, O., Glück, R., and Thiemann, P. (eds.): *Partial Evaluation: International Seminar*. Lecture Notes in Computer Science, Vol. 1110. Springer-Verlag, Berlin Heidelberg New York (1996) 183–251.
(<http://www.cs.chalmers.se/~rjmh/Papers/typed-pe.ps>)
12. Hughes, J.: An introduction to program specialization by type inference. In: *Glasgow Workshop on Functional Programming*. Glasgow University (1996).
(<http://www.cs.chalmers.se/~rjmh/Papers/glasgow-96.dvi>)
13. Hughes, J.: The correctness of type specialization. In: Smolka, G. (ed.): *9th European Symposium on Programming: ESOP '00*. Lecture Notes in Computer Science, Vol. 1782. Springer-Verlag, Berlin Heidelberg New York, 215–229
14. Jones, N. D. et al.: Challenging problems in partial evaluation and mixed computation. In: Bjørner, D., Ershov, A. P., and Jones, N. D. (eds.): *Partial Evaluation and Mixed Computation: IFIP TC2 Workshop*. North-Holland, Amsterdam, The Netherlands (1987) 1–14. Also pages 291–303 of [3]
15. Jones, N. D., Gomard, C. K., and Sestoft, P.: *Partial Evaluation and Automatic Program Generation*. Prentice Hall, Englewood Cliff, NJ, USA (1993).
(<http://www.dina.kvl.dk/~sestoft/pebook/pebook.html>)
16. Launchbury, J.: A strongly-typed self-applicable partial evaluator. In: Hughes, J. (ed.): *Functional Programming Languages and Computer Architecture: 5th ACM Conference, FPCA '91*. Lecture Notes in Computer Science, Vol. 523. Springer-Verlag, Berlin Heidelberg New York, 145–164
17. Mogensen, T. Æ.: Constructor specialization. In: Schmidt, D. (ed.): *Partial Evaluation and Semantics-Based Program Manipulation: ACM SIGPLAN Symposium, PEPM '93*. 22–32
18. Romanenko, S. and Sestoft, P.: Moscow ML version 1.44. A light-weight implementation of Standard ML (1999). (<http://www.dina.kvl.dk/~sestoft/mosml.html>)
19. Taha, W. and Makhholm, H.: Tag elimination; or, Type specialisation is a type-indexed effect. In: *Subtyping & Dependent Types in Programming: APPSEM Workshop, DTP '00*. INRIA technical report.
(<http://www-sop.inria.fr/oasis/DTP00/Proceedings/taha.ps>)
20. Welinder, M.: *Partial Evaluation and Correctness*. PhD thesis, Department of Computer Science, University of Copenhagen (1998). DIKU-TR-98/13