BRICS RS-95-41 Danvy et al.: Eta-Expansion Does The Trick

Matematisk Institut
Aarhus Universitet
Trykkeriet

BRICS

Basic Research in Computer Science

Eta-Expansion Does The Trick

Olivier Danvy
Karoline Malmkjær
Jens Palsberg

BRICS Report Series RS-95-41

ISSN 0909-0878 August 1995

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

http://www.brics.aau.dk/BRICS/
ftp ftp.brics.aau.dk (cd pub/BRICS)

Eta-Expansion Does The Trick

Olivier Danvy* Karoline Malmkjær* Jens Palsberg†
Aarhus University MIT

August 10, 1995

Abstract

Partial-evaluation folklore has it that massaging one's source programs can make them specialize better. In Jones, Gomard, and Sestoft's recent textbook, a whole chapter is dedicated to listing such "binding-time improvements": non-standard use of continuation-passing style, eta-expansion, and a popular transformation called "The Trick". We provide a unified view of these binding-time improvements, from a typing perspective.

Just as a proper treatment of product values in partial evaluation requires partially static values, a proper treatment of disjoint sums requires moving static contexts across dynamic case expressions. This requirement precisely accounts for the non-standard use of continuation-passing style encountered in partial evaluation. In this setting, eta-expansion acts as a uniform binding-time coercion between values and contexts, be they of function type, product type, or disjoint-sum type. For the latter case, it achieves "The Trick".

In this paper, we extend Gomard and Jones's partial evaluator for the λ-calculus, λ-Mix, with products and disjoint sums; we point out how eta-expansion for disjoint sums does The Trick; we generalize our earlier work by identifying that eta-expansion can be obtained in the binding-time analysis simply by adding two coercion rules; and we specify and prove the correctness of our extension to λ-Mix.

Keywords: Partial evaluation, binding-time analysis.

*Computer Science Department, Aarhus University, Ny Munkegade, DK-8000 Aarhus C, Denmark; E-mail: {danvy,karoline}@daimi.aau.dk.

†Laboratory for Computer Science, Massachusetts Institute of Technology, NE43-340, 545 Technology Square, Cambridge, Massachusetts 02139, USA; E-mail: palsberg@theory.lcs.mit.edu.

1 Introduction

Partial evaluation is a program-transformation technique for specializing programs [8, 16]. Modern partial evaluators come in two flavors: online and offline. An online partial evaluator specializes program in an interpretive way [21, 25]. An offline partial evaluator is divided into two stages:

1. a *binding-time analysis* determining which parts of the source program are known (the "static" parts) and which parts may not be known (the "dynamic" parts);

2. a *program specializer* reducing the static parts and reconstructing the dynamic parts, thus producing the residual program.

The two stages must fit together such that (1) no static parts are left in the residual program, and (2) no static computation depends on the result of a dynamic computation [14, 18, 19].

In this paper, we consider offline partial evaluation, but our results also apply to online partial evaluation.

In an offline partial evaluator, the precision of the binding-time analysis determines the effectiveness of the program specializer [8, 16]. Informally, the more parts of a source program are classified to be static by the binding-time analysis, the more parts are processed away by the specializer.

Practical experience with partial evaluation shows that users need to massage their source programs to make binding-time analysis classify more program parts as static, and thus to make specialization yield better results. Jones, Gomard, and Sestoft's textbook [16, Chapter 12] documents three such "binding-time improvements": continuation-passing style, eta-expansion, and "The Trick".

1.1 Continuation-passing style

For some expressions, evaluation reduces to evaluating sub-expressions; for example, evaluating a let expression reduces to evaluating its body, and evaluating a conditional expression reduces to evaluating one of the conditional branches. Classifying such outer expressions as dynamic forces these inner expressions to be dynamic as well, even when they are actually static and the context of the outer expression, given a static value, could be classified as static. For example, in terms such as

$$10 + (\text{let } x = D \text{ in } 2 \text{ end})$$

$$10 + (\text{case } D \text{ of } \text{inleft}(t_1) \Rightarrow 1 \mid \text{inright}(t_2) \Rightarrow 2 \text{ end})$$

if D is dynamic, both the let and the case expressions need to be reconstructed. (In the presence of computational effects, e.g., divergence, unfolding such a let expression statically is unsafe.) The second argument of + is therefore dynamic, and thus + is classified to be dynamic as well, even though at run time, both expressions reduce to a value that could have been computed at specialization time. Against this backdrop, moving the context $[10 + [\cdot]]$ inside the let and the case expressions makes it possible to classify + to be static and thus to compute the addition at specialization time. This context move can be achieved either by a source transformation such as the CPS transformation or by delimiting the "static" continuation of the specializer and relocating it inside the reconstructed expression. Both methods are documented in the literature [5, 7, 16, 17]. Note that this change in the specializer requires a corresponding change in the binding-time analysis.

1.2 Eta-expansion

Jones, Gomard, and Sestoft list eta-expansion as an effective binding-time improvement [16]. In an earlier work [10], we showed that a source eta-expansion serves as a binding-time coercion for static higher-order values in dynamic contexts and for dynamic values in potentially static contexts expecting higher-order values (see Section 3.1). We proposed and proved the correctness of a binding-time analysis that generates these binding-time coercions at points of conflict, instead of taking the conservative solution of dynamizing both values and contexts.

In the same paper [10], we also pointed out that an analog problem occurs for products, and that the analog of eta-expansion for products serves as a binding-time coercion for static product values in dynamic contexts and for dynamic values in potentially static contexts expecting product values (see Section 3.2). We did not, however, present the corresponding binding-time analysis generating these binding-time coercions at points of conflict, nor did we conside disjoint sums.

In summary, eta-redexes provide a syntactic representation of binding-time coercions, either from static to dynamic, or from dynamic to static.

1.3 "The Trick"

In their partial-evaluation textbook [16], Jones, Gomard, and Sestoft document a folklore binding-time improvement, referring to it as "The Trick".

The Trick has not been formalized. Intuitively, it is used to process dynamic choices of static values, *i.e.*, when finitely many static values may occur in a dynamic context. Enumerating these values makes it possible to plug each of them into the context, thereby turning it into a static context and enabling more static computation.

The Trick can also be used on any finite type, such as booleans or characters, by enumerating its elements. Alternatively, one may wish to cut on the number of static possibilities that can be encountered at a program point — for example, only finitely many characters (instead of the whole alphabet) may occur in a regular-expression interpreter [16, Section 12.2]. The Trick is usually carried out explicitly by the programmer (see the while loop in Jones and Gomard's Imperative Mix [16, Section 4.8.3]).

This enumeration of static values could also be obtained by program analysis. Exploiting the results of such a program analysis would make it possible to automatize The Trick. In fact, a program analysis determining finite ranges of values that may occur at a program point does enable The Trick. For example, control-flow analysis [23] (also known as closure analysis [22]) determines a conservative approximation of which λ-abstractions can give rise to a closure that may occur at an application site. The application site can be transformed into a case-expression listing all the possible λ-abstractions and performing a first-order call to the corresponding λ-abstraction in each branch. This defunctionalization technique was proposed by Reynolds in the early seventies [20]. Since the end of the eighties, it is used by such partial evaluators as Similix to handle higher-order programs [4]. The conclusion of this is that Jones, Gomard, and Sestoft actually do use an automated version of The Trick [16, Section 10.1.4, Item (1)], even if they do not present it as such.

In summary, and according to the literature, The Trick appears as yet another powerful binding-time improvement. It has not been formalized.

1.4 This paper

In this paper we present and prove the correctness of a partial evaluator that both automates and unifies the binding-time improvements listed above. Section 2 presents an extension of λ-Mix which handles products and disjoint sums properly. Section 3 illustrates the effect of eta-expansion in this continuation-based partial evaluator. In particular, eta-expansion of disjoint-sums values does The Trick. Section 4 extends the binding-time analysis of Section 2 with coercions as eta-redexes. Section 5 proves the

correctness of this extended partial evaluator. Section 6 assesses our results and Section 7 concludes.

1.5 Notation

Consistently with Nielson and Nielson [18], we use overlining to denote "static" and underlining to denote "dynamic". For purposes of annotation, we use "@" (pronounced "apply") to denote applications, and we abbreviate $(e_0 @ e_1) @ e_2$ by $e_0 @ e_1 @ e_2$, and $e_0 @ (\lambda x.e)$ by $e_0 @ \lambda x.e$.

A context is an expression with one hole [2].

We assume Barendregt's Variable Convention [2]: when a λ-term occurs in this article, all bound variables are chosen to be different from the free variables. This can be achieved by renaming bound variables.

Eta-expanding a higher-order expression $e : \tau_1 \to \tau_2$ yields the expression

$$\lambda v.e @ v$$

where v does not occur free in e [2]. By analogy, "eta-expanding" a product expression $e : \tau_1 \times \tau_2$ yields the expression

$$\text{pair}(\text{fst } e, \text{snd } e)$$

and "eta-expanding" a disjoint-sum expression $e : \tau_1 + \tau_2$ yields the expression

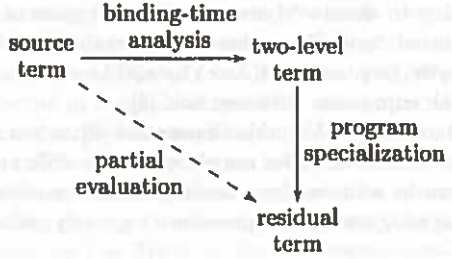
$$\text{case } e \text{ of inleft}(x_1) \Rightarrow \text{inleft}(x_1) \mid \text{inright}(x_2) \Rightarrow \text{inright}(x_2) \text{ end.}$$

2 An Extension of λ-Mix handling Products and Disjoint Sums

Our starting point is Gomard and Jones's partial evaluator λ-Mix, an offline partial evaluator for the λ-calculus [11, 12, 16]. We extend it to handle products and disjoint sums. Like Gomard and Jones's, our binding-time analysis is monovariant in that it associates one binding-time type for each source expression.

Our partial evaluator provides a proper treatment of disjoint sums, where a dynamic sum of two static values is *not* approximated to be dynamic if its context of use is static. Instead, this context is duplicated during specialization. Bondorf has given a specification of this technique, but no proof of correctness [5].

Figure 1 displays the syntax of a λ-calculus with products and disjoint sums. A binding-time analysis (Section 2.1) maps a λ-term into a two-level λ-term (Figure 2). Program specialization (Section 2.2) reduces all the static parts of a two-level λ-term and yields a completely dynamic λ-term. Erasing its annotations yields the residual, specialized λ-term.



2.1 Binding-time analysis

Figure 3 displays Gomard's binding-time analysis, restricted to the pure λ-calculus. Types are finite and generated from the grammar of Figure 2. The type d denotes the type of dynamic values. The judgement

$$A \vdash e : t \triangleright w$$

should be read as follows: under hypothesis A, the λ-term e can be assigned the type t with the annotated term w.

Figures 4 and 5 display the extension of this binding-time analysis to products and disjoint sums. The extension to products is straightforward. In the extension to disjoint sums, the binding time of a case expression is independent of the binding time of its test, *even when this test is dynamic.*

2.2 Program specialization

Program specialization reduces the static parts of a two-level λ-term. The specification of program specialization has the form of an operational semantics. If e and e' are two-level λ-terms, then $e \longrightarrow e'$ means that e reduces to e'. Figure 6 displays the three basic evaluation rules, and Figure 7 displays four novel "code-motion" rules. Each code-motion rule duplicates the static context of a dynamic case-expression and moves the copies to the branches of the case-expression. This creates new redexes, which fits together with

$$\begin{array}{rcl} e & ::= & x \mid \\ & & \lambda x.e \mid e_0@e_1 \mid \mathrm{pair}(e_1, e_2) \mid \mathrm{fst}\ e \mid \mathrm{snd}\ e \mid \\ & & \mathrm{inleft}(e) \mid \mathrm{inright}(e) \mid \\ & & \mathrm{case}\ e\ \mathrm{of}\ \mathrm{inleft}(x_1) \Rightarrow e_1 \mid \mathrm{inright}(x_2) \Rightarrow e_2\ \mathrm{end} \end{array}$$

Figure 1: BNF of the λ-calculus

$$\begin{array}{rcl} \tau & ::= & d \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \\ e & ::= & x \mid \\ & & \overline{\lambda} x.e \mid e_0 \overline{@} e_1 \mid \overline{\mathrm{pair}}(e_1, e_2) \mid \overline{\mathrm{fst}}\ e \mid \overline{\mathrm{snd}}\ e \mid \\ & & \underline{\lambda} x.e \mid e_0 \underline{@} e_1 \mid \underline{\mathrm{pair}}(e_1, e_2) \mid \underline{\mathrm{fst}}\ e \mid \underline{\mathrm{snd}}\ e \mid \\ & & \overline{\mathrm{inleft}}(e) \mid \overline{\mathrm{inright}}(e) \mid \\ & & \overline{\mathrm{case}}\ e\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mid \overline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \overline{\mathrm{end}} \mid \\ & & \underline{\mathrm{inleft}}(e) \mid \underline{\mathrm{inright}}(e) \mid \\ & & \underline{\mathrm{case}}\ e\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}} \end{array}$$

Figure 2: BNF of the two-level λ-calculus

$$A \vdash x : A(x) \triangleright x$$

$$\frac{A[x \mapsto \tau_1] \vdash e : \tau_2 \triangleright w}{A \vdash \lambda x.e : \tau_1 \to \tau_2 \triangleright \overline{\lambda} x.w} \qquad \frac{A[x \mapsto d] \vdash e : d \triangleright w}{A \vdash \lambda x.e : d \triangleright \underline{\lambda} x.w}$$

$$\frac{A \vdash e_0 : \tau_1 \to \tau_2 \triangleright w_0 \qquad A \vdash e_1 : \tau_1 \triangleright w_1}{A \vdash e_0@e_1 : \tau_2 \triangleright w_0 \overline{@} w_1}$$

$$\frac{A \vdash e_0 : d \triangleright w_0 \qquad A \vdash e_1 : d \triangleright w_1}{A \vdash e_0@e_1 : d \triangleright w_0 \underline{@} w_1}$$

Figure 3: Gomard's binding-time analysis for the pure λ-calculus

$$\frac{A \vdash e_1 : \tau_1 \triangleright w_1 \qquad A \vdash e_2 : \tau_2 \triangleright w_2}{A \vdash \mathrm{pair}(e_1, e_2) : \tau_1 \times \tau_2 \triangleright \overline{\mathrm{pair}}(w_1, w_2)}$$

$$\frac{A \vdash e_1 : d \triangleright w_1 \qquad A \vdash e_2 : d \triangleright w_2}{A \vdash \mathrm{pair}(e_1, e_2) : d \triangleright \underline{\mathrm{pair}}(w_1, w_2)}$$

$$\frac{A \vdash e : \tau_1 \times \tau_2 \triangleright w}{A \vdash \mathrm{fst}\, e : \tau_1 \triangleright \overline{\mathrm{fst}}\, w} \qquad \frac{A \vdash e : \tau_1 \times \tau_2 \triangleright w}{A \vdash \mathrm{snd}\, e : \tau_2 \triangleright \overline{\mathrm{snd}}\, w}$$

$$\frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{fst}\, e : d \triangleright \underline{\mathrm{fst}}\, w} \qquad \frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{snd}\, e : d \triangleright \underline{\mathrm{snd}}\, w}$$

Figure 4: Extension of Gomard's binding-time analysis to products

$$\frac{A \vdash e : \tau_1 + \tau_2 \triangleright w \quad A[x_1 \mapsto \tau_1] \vdash e_1 : \tau \triangleright w_1 \quad A[x_2 \mapsto \tau_2] \vdash e_2 : \tau \triangleright w_2}{\begin{array}{ll} A \vdash \mathrm{case}\, e\, \mathrm{of} & : \tau \triangleright \overline{\mathrm{case}}\, w\, \overline{\mathrm{of}} \\ \quad \mathrm{inleft}(x_1) \Rightarrow e_1 & \quad \overline{\mathrm{inleft}}(x_1) \Rightarrow w_1 \\ \quad \mid \mathrm{inright}(x_2) \Rightarrow e_2 & \quad \mid \overline{\mathrm{inright}}(x_2) \Rightarrow w_2 \\ \quad \mathrm{end} & \quad \overline{\mathrm{end}} \end{array}}$$

$$\frac{A \vdash e : d \triangleright w \quad A[x_1 \mapsto d] \vdash e_1 : \tau \triangleright w_1 \quad A[x_2 \mapsto d] \vdash e_2 : \tau \triangleright w_2}{\begin{array}{ll} A \vdash \mathrm{case}\, e\, \mathrm{of} & : \tau \triangleright \underline{\mathrm{case}}\, w\, \underline{\mathrm{of}} \\ \quad \mathrm{inleft}(x_1) \Rightarrow e_1 & \quad \underline{\mathrm{inleft}}(x_1) \Rightarrow w_1 \\ \quad \mid \mathrm{inright}(x_2) \Rightarrow e_2 & \quad \mid \underline{\mathrm{inright}}(x_2) \Rightarrow w_2 \\ \quad \mathrm{end} & \quad \underline{\mathrm{end}} \end{array}}$$

$$\frac{A \vdash e : \tau_1 \triangleright w}{A \vdash \mathrm{inleft}(e) : \tau_1 + \tau_2 \triangleright \overline{\mathrm{inleft}}(w)} \qquad \frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{inleft}(e) : d \triangleright \underline{\mathrm{inleft}}(w)}$$

$$\frac{A \vdash e : \tau_2 \triangleright w}{A \vdash \mathrm{inright}(e) : \tau_1 + \tau_2 \triangleright \overline{\mathrm{inright}}(w)} \qquad \frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{inright}(e) : d \triangleright \underline{\mathrm{inright}}(w)}$$

Figure 5: Extension of Gomard's binding-time analysis to sums

Static applications:

$$(\overline{\lambda} x.e)\overline{@}e_1 \longrightarrow e[e_1/x]$$

Static decompositions:

$$\overline{\mathrm{fst}}\ \overline{\mathrm{pair}}(e_1, e_2) \longrightarrow e_1 \qquad \overline{\mathrm{snd}}\ \overline{\mathrm{pair}}(e_1, e_2) \longrightarrow e_2$$

Static projections:

$$\begin{array}{ll} \overline{\mathrm{case}}\ \overline{\mathrm{inleft}}(e)\ \overline{\mathrm{of}} & \longrightarrow e_1[e/x_1] \\ \quad \overline{\mathrm{inleft}}(x_1) \Rightarrow e_1 & \\ \quad \mid \overline{\mathrm{inright}}(x_2) \Rightarrow e_2 & \\ \quad \overline{\mathrm{end}} & \end{array} \qquad \begin{array}{ll} \overline{\mathrm{case}}\ \overline{\mathrm{inright}}(e)\ \overline{\mathrm{of}} & \longrightarrow e_2[e/x_2] \\ \quad \overline{\mathrm{inleft}}(x_1) \Rightarrow e_1 & \\ \quad \mid \overline{\mathrm{inright}}(x_2) \Rightarrow e_2 & \\ \quad \overline{\mathrm{end}} & \end{array}$$

Figure 6: Operational semantics of the two-level λ-calculus — evaluation rules

the rule for binding-time analysis of case-expressions of Figure 5. For each evaluation context $E[\cdot]$, if $e \longrightarrow e'$, then $E[e] \longrightarrow E[e']$.

Our extension of λ-Mix is correct, as proven in Section 5.

3 Examples

We first briefly summarize how eta-expansion works for functions and products, and then we give two examples of how our partial evaluator does The Trick.

3.1 Coercions for functions

As illustrated in our earlier work [10], for functions, eta-expansion is useful in two cases. The first is where a dynamic context $[\cdot]$, expecting a higher-order value of type d (one could be tempted to write "of type $\tau_1 \underline{\to} \tau_2$"), can be coerced into a static context

$$\underline{\lambda} v.[\cdot]\overline{@}v$$

that expects a value of type $\tau_1 \overline{\to} \tau_2$. The second useful case is where a dynamic higher-order value e of type d (again, one could be tempted to

Static applications:

$$(\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow e_2\ \underline{\text{end}})\overline{@}e \longrightarrow$$
$$\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow e_1\overline{@}e \mid \underline{\text{inright}}(x_2) \Rightarrow e_2\overline{@}e\ \underline{\text{end}}$$

Static decompositions:

$$\overline{\text{fst}}\ (\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow e_2\ \underline{\text{end}}) \longrightarrow$$
$$\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow \overline{\text{fst}}\ e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow \overline{\text{fst}}\ e_2\ \underline{\text{end}}$$

$$\overline{\text{snd}}\ (\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow e_2\ \underline{\text{end}}) \longrightarrow$$
$$\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow \overline{\text{snd}}\ e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow \overline{\text{snd}}\ e_2\ \underline{\text{end}}$$

Static projections:

$$\overline{\text{case}}\ (\underline{\text{case}}\ e_0\ \underline{\text{of}}\ \underline{\text{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\text{inright}}(x_2) \Rightarrow e_2\ \underline{\text{end}})\ \overline{\text{of}}$$
$$\overline{\text{inleft}}(x_1') \Rightarrow e_1'$$
$$\mid \overline{\text{inright}}(x_2') \Rightarrow e_2'$$
$$\overline{\text{end}}$$
$$\longrightarrow \underline{\text{case}}\ e_0\ \underline{\text{of}}$$
$$\underline{\text{inleft}}(x_1) \Rightarrow \overline{\text{case}}\ e_1\ \overline{\text{of}}\ \overline{\text{inleft}}(x_1') \Rightarrow e_1' \mid \overline{\text{inright}}(x_2') \Rightarrow e_2'\ \overline{\text{end}}$$
$$\mid \underline{\text{inright}}(x_2) \Rightarrow \overline{\text{case}}\ e_2\ \overline{\text{of}}\ \overline{\text{inleft}}(x_1') \Rightarrow e_1' \mid \overline{\text{inright}}(x_2') \Rightarrow e_2'\ \overline{\text{end}}$$
$$\underline{\text{end}}$$

Figure 7: Operational semantics of the two-level λ-calculus — code-motion rules

write $\tau_1 \underline{\rightarrow} \tau_2$) can be coerced into a static value

$$\overline{\lambda} v.e\underline{@}v$$

of type $\tau_1 \overline{\rightarrow} \tau_2$ that will fit into a static context.

3.2 Coercions for products

A similar situation occurs for partially static values: whenever such a value occurs in a dynamic context, the value is dynamized, and conversely, whenever a partially static context receives a dynamic value, the context is dynamized as well. Let us consider pairs. A static pair p of type $d\overline{\times}d$ can be coerced to

$$\underline{\text{pair}}(\overline{\text{fst}}\ p, \overline{\text{snd}}\ p)$$

which has type d. A dynamic pair p of type d can be coerced to

$$\overline{\text{pair}}(\underline{\text{fst}}\ p, \underline{\text{snd}}\ p)$$

which has type $d\overline{\times}d$.

For example, if the following expression occurs in a dynamic context

$$\text{fst}\ e$$

where e has type $d \times d$, the result of binding-time analysis

$$\underline{\text{fst}}\ e$$

where e has type d. If we eta-expand the result, it will read:

$$\underline{\text{fst}}\ (\underline{\text{pair}}(\overline{\text{fst}}\ e, \overline{\text{snd}}\ e)).$$

This term has type d, which matches the type of its context, and the partially static pair e remains partially static, thanks to the coercion.

Conversely, if the value of two expressions e (of type d) and e' (of type $d \times d$) can occur in the same context, binding-time analysis classifies e' to be dynamic and, as a byproduct, dynamizes this context. Again, e could be eta-expanded to read

$$\overline{\text{pair}}(\underline{\text{fst}}\ e, \underline{\text{snd}}\ e).$$

This term has type $d \times d$, which avoids to dynamize the context and thus makes it possible to keep e' a static pair, thanks to the coercion.

3.3 Coercions for disjoint sums

The same coercions apply to disjoint sums. In the following, we give two examples of how The Trick can be achieved by eta-expansion in the presence of our new rules for binding-time analysis and transformation of case-expressions.

3.3.1 Static injection in a dynamic context

The following expression is partially evaluated in a context where f is dynamic.

$$(\lambda v.f\underline{@}(\text{case}\ v\ \text{of inleft}(a) \Rightarrow a + 20 \mid \text{inright}(b) \Rightarrow \ldots\ \text{end})\underline{@}v)\underline{@}\text{inleft}(10)$$

Assume this β-redex will be reduced. Notice that v occurs twice: once as the test part of a case expression, and once as the argument of the application of f to the case expression. Since f is dynamic, its application is dynamic, and the application of that expression is dynamic as well. Thus the binding-time analysis classifies v to be dynamic since it occurs in a dynamic context, and in turn both the case expression and inleft(10) are also classified as dynamic. Overall, binding-time analysis yields the following two-level term.

$$(\overline{\lambda}v.f\underline{@}(\underline{\text{case}}\ v\ \underline{\text{of}}\ \underline{\text{inleft}}(a) \Rightarrow a \underline{+} 20 \mid \underline{\text{inright}}(b) \Rightarrow \ldots\ \underline{\text{end}})\underline{@}v)\overline{@}\underline{\text{inleft}}(10)$$

In this term, f has type d and v also has type d.

After specialization (*i.e.*, reduction of static expressions and reconstruction of dynamic expressions), the residual term reads as follows.

$$f@(\text{case inleft}(10)\ \text{of inleft}(a) \Rightarrow a + 20 \mid \text{inright}(b) \Rightarrow \ldots\ \text{end})@\text{inleft}(10)$$

The fact that inleft(10), a partially static value, occurs in the dynamic context $f@(\text{case } v \text{ of inleft}(a) \Rightarrow a + 20 \mid \text{inright}(b) \Rightarrow \ldots \text{ end})@[\cdot]$ "pollutes" its occurrence in the potentially static context case $[\cdot]$ of inleft$(a) \Rightarrow a + 20 \mid$ inright$(b) \Rightarrow \ldots$ end, so that neither is reduced statically.

NB: Since v is dynamic and occurs twice, a cautious binding-time analysis would reclassify the outer application to be dynamic: there is usually no point in duplicating residual code. In that case, the expression is totally dynamic and so is not simplified at all.

In this situation, a binding-time improvement is possible since inleft(10) will occur in a dynamic context. We can coerce this occurrence by eta-expanding the dynamic context (the eta-redex is boxed).

$$\begin{array}{l}\lambda v.f@(\text{case } v \text{ of inleft}(a) \Rightarrow a + 20 \mid \text{inright}(b) \Rightarrow \ldots \text{ end}) \\ \quad @ \\ \quad \boxed{\text{case } v \text{ of inleft}(a) \Rightarrow \text{inleft}(a) \mid \text{inright}(b) \Rightarrow \text{inright}(b)\ \text{end}} \\ @ \\ \text{inleft}(10)\end{array}$$

Binding-time analysis now yields the following two-level term.

$$\begin{array}{l}\overline{\lambda}v.f\underline{@}(\overline{\text{case}}\ v\ \overline{\text{of}}\ \overline{\text{inleft}}(a) \Rightarrow a \overline{+} 20 \mid \overline{\text{inright}}(b) \Rightarrow \ldots\ \overline{\text{end}}) \\ \quad \underline{@} \\ \quad \overline{\text{case}}\ v\ \overline{\text{of}}\ \overline{\text{inleft}}(a) \Rightarrow \underline{\text{inleft}}(a) \mid \overline{\text{inright}}(b) \Rightarrow \underline{\text{inright}}(b)\ \overline{\text{end}} \\ \overline{@} \\ \overline{\text{inleft}}(10)\end{array}$$

Here, v is not approximated to be dynamic: it has the type Int$+t$, for some t.

Specialization yields the residual term

$$f@30@\text{inleft}(10)$$

which is more reduced statically.

Let us now illustrate the dual case, where a dynamic injection in a potentially static context dynamizes this context.

3.3.2 Dynamic injection in a static context

The following expression is partially evaluated in a context where d is dynamic.

$$\begin{array}{l}\lambda f.\ \ldots\ f@d\ \ldots\ f@\text{inleft}(\lambda x.x)\ \ldots \\ @ \\ \lambda v.\text{case } v \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow \ldots \text{ end}\end{array}$$

Assume this β-redex will be reduced. Notice that f occurs twice: it is applied both to a static value and to a dynamic value. A monovariant binding-time analysis (see Figures 3, 4, and 5) thus would approximate its argument to be dynamic and would yield the following two-level term.

$$\begin{array}{l}\overline{\lambda}f.\ \ldots\ f\underline{@}d\ \ldots\ f\underline{@}\text{inleft}(\lambda x.x)\ \ldots \\ \overline{@} \\ \underline{\lambda}v.\underline{\text{case}}\ v\ \underline{\text{of}}\ \underline{\text{inleft}}(a) \Rightarrow a\underline{@}10 \mid \underline{\text{inright}}(b) \Rightarrow \ldots\ \underline{\text{end}}\end{array}$$

In this term, f has type d.

Specialization yields the following residual term.

$$\begin{array}{l}\ldots \\ (\lambda v.\text{case } v \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow \ldots \text{ end})@d \\ \ldots \\ (\lambda v.\text{case } v \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow \ldots \text{ end})@\text{inleft}(\lambda x.x) \\ \ldots\end{array}$$

The fact that d, a dynamic value, occurs in the potentially static context $f@[\cdot]$ "dynamizes" this context, which in turn, dynamizes inleft$(\lambda x.x)$.

In this situation, a binding-time improvement is possible to make inleft$(\lambda x.x)$ occur in a static context always. We can coerce the bother-

ing occurrence of d by eta-expanding it (the eta-redex is boxed).

$\lambda f.\ ...$
$\quad f@\boxed{\text{case } d \text{ of inleft}(a) \Rightarrow \text{inleft}(a) \mid \text{inright}(b) \Rightarrow \text{inright}(b) \text{ end}}$
$\quad ...$
$\quad f@\text{inleft}(\lambda x.x)$
$\quad ...$
@
$\lambda v.\text{case } v \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow ... \text{ end}$

This eta-expansion does The Trick. Even though d is not statically known, its type tells us that it is either some dynamic value a or some dynamic value b. Program specialization automatically plugs these values into the enclosing context (see Figure 7).

But this is not enough because now $\lambda x.x$ will be dynamized by the newly introduced occurrence of a. Indeed, binding-time analysis yields the following two-level term.

$\overline{\lambda} f.\ ...$
$\quad f\overline{@}\underline{\text{case}}\ d\ \underline{\text{of}}\ \underline{\text{inleft}}(a) \Rightarrow \overline{\text{inleft}}(a) \mid \underline{\text{inright}}(b) \Rightarrow \overline{\text{inright}}(b)\ \underline{\text{end}}$
$\quad ...$
$\quad f\overline{@\text{inleft}}(\underline{\lambda} x.x)$
$\quad ...$
$\overline{@}$
$\overline{\lambda} v.\overline{\text{case}}\ v\ \overline{\text{of}}\ \overline{\text{inleft}}(a) \Rightarrow a\underline{@}10 \mid \overline{\text{inright}}(b) \Rightarrow ...\ \overline{\text{end}}$

In this term, f has type $d \to d$.

Specialization moves the context of the dynamic case expression in each of its branches and produces the following residual term.

$\quad ...$
$\quad \text{case } d \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow ... \text{ end}$
$\quad ...$
$\quad (\lambda x.x)@10$
$\quad ...$

The fact that a, a dynamic value, occurs in the potentially static context $[\cdot]@10$ dynamizes this context, which in turns, dynamizes $\lambda x.x$.

Fortunately, we already solved that problem in Section 3.1, using eta-expansion. The new eta-redex is boxed.

$\lambda f.\ ...$
$\quad f@\text{case } d \text{ of inleft}(a) \Rightarrow \text{inleft}(\boxed{\lambda z.a@z}) \mid \text{inright}(b) \Rightarrow \text{inright}(b) \text{ end}$
$\quad ...$
$\quad f@\text{inleft}(\lambda x.x)$
$\quad ...$
@
$\lambda v.\text{case } v \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow ... \text{ end}$

Binding-time analysis now yields the following two-level term.

$\overline{\lambda} f.\ ...$
$\quad f\overline{@}\underline{\text{case}}\ d\ \underline{\text{of}}\ \underline{\text{inleft}}(a) \Rightarrow \overline{\text{inleft}}(\overline{\lambda} z.a\underline{@}z) \mid \underline{\text{inright}}(b) \Rightarrow \overline{\text{inright}}(b)\ \underline{\text{end}}$
$\quad ...$
$\quad f\overline{@\text{inleft}}(\overline{\lambda} x.x)$
$\quad ...$
$\overline{@}$
$\overline{\lambda} v.\overline{\text{case}}\ v\ \overline{\text{of}}\ \overline{\text{inleft}}(a) \Rightarrow a\overline{@}10 \mid \overline{\text{inright}}(b) \Rightarrow ...\ \overline{\text{end}}$

Here, f has type $((d \to d) + t) \to d$, for some t. Thus neither $\text{inleft}(\lambda x.x)$ nor $\lambda x.x$ are approximated to be dynamic.

Specialization yields the following residual term.

$\quad ...$
$\quad (\text{case } d \text{ of inleft}(a) \Rightarrow a@10 \mid \text{inright}(b) \Rightarrow ... \text{ end})$
$\quad ...$
$\quad 10$
$\quad ...$

This residual term is more reduced statically.

3.3.3 Conclusions

For functions, products, and disjoint sums, eta-redexes act as binding-time coercions. Also, and as illustrated in the last example, they synergize. In particular, the first eta-expansion of Section 3.3.2 corresponds to applying The Trick. Even though d is unknown, its type tells us that it can be either some (dynamic) value a or b. Program specialization automatically plugs these values into the surrounding context (see Figure 7).

$$\frac{A \vdash e : d \triangleright w \qquad \tau \vdash z \Rightarrow m \qquad \emptyset[z \mapsto d] \vdash m : \tau \triangleright w'}{A \vdash e : \tau \triangleright w'[w/z]}$$

$$\frac{A \vdash e : \tau \triangleright w \qquad \tau \vdash z \Rightarrow m \qquad \emptyset[z \mapsto \tau] \vdash m : d \triangleright w'}{A \vdash e : d \triangleright w'[w/z]}$$

Figure 8: Extension of Gomard's binding-time analysis to binding-time coercions

$$d \vdash e \Rightarrow e$$

$$\frac{\tau_1 \vdash x \Rightarrow x' \qquad \tau_2 \vdash e@x' \Rightarrow e'}{\tau_1 \to \tau_2 \vdash e \Rightarrow \lambda x.e'}$$

$$\frac{\tau_1 \vdash \mathrm{fst}\ e \Rightarrow e_1 \qquad \tau_2 \vdash \mathrm{snd}\ e \Rightarrow e_2}{\tau_1 \times \tau_2 \vdash e \Rightarrow \mathrm{pair}(e_1, e_2)}$$

$$\frac{\tau_1 \vdash x_1 \Rightarrow e_1 \qquad \tau_2 \vdash x_2 \Rightarrow e_2}{\tau_1 + \tau_2 \vdash e \Rightarrow \mathrm{case}\ e\ \mathrm{of\ inleft}(x_1) \Rightarrow e_1 \mid \mathrm{inright}(x_2) \Rightarrow e_2\ \mathrm{end}}$$

Figure 9: Type-directed eta expansion

4 Binding-Time Analysis with Eta-Expansion

In an earlier work [10], we proposed and proved the correctness of a binding-time analysis that generates binding-time coercions for higher-order values at points of conflict, instead of taking the conservative solution of dynamizing both values and contexts. We pointed out the analogous need for binding-time coercions for products, but did not present the corresponding binding-time analysis generating these binding-time coercions at points of conflict. This binding-time analysis can be obtained by extending the binding-time analysis of Figures 3, 4, and 5 with Figures 8 and 9.

Figure 8 displays two general eta-expansion rules. Intuitively, the two rules can be understood as being able (1) to coerce the binding-time type d to any type τ, and (2) to coerce any type τ to the type d.

Eta-expansion itself is defined in Figure 9. It is type-directed, and thus it can insert several embedded eta-redexes in a way that is reminiscent of Berger and Schwichtenberg's normalization of λ-terms [3, 9].

With this new binding-time analysis, all the examples of Section 3 now specialize well without binding-time improvement. In particular, no tricks are required from the partial-evaluation user — they were a tell-tale of too coarse binding-time coercions in existing binding-time analyses.

5 Correctness

We now state and prove that our binding-time analysis is correct with respect to the operational semantics of 2-level λ-terms. The statement of correctness is taken from Palsberg [19] and Wand [24], who proved correctness of two other binding-time analyses. The proof techniques are well-known; we omit the details.

If w is a 2-level λ-term, then $\widehat{w}$ denotes the underlying λ-term.

We first prove that if e can be annotated as w, then so can $\widehat{w}$. This enables us to simplify the statements and proofs of subsequent theorems.

Theorem 5.1 (Simplification) *If $A \vdash e : \tau \triangleright w$, then $A \vdash \widehat{w} : \tau \triangleright w$.*

Proof. By induction on the structure of the derivation of $A \vdash e : \tau \triangleright w$. □

We then prove subject reduction, using a substitution lemma.

Lemma 5.2 (Substitution)

If $A \vdash \widehat{w_1} : \tau \triangleright w_1$ and $A' \vdash \widehat{w_2} : \tau' \triangleright w_2$,
then $A'' \vdash \widehat{w_2}[\widehat{w_1}/z] : \tau' \triangleright w_2[w_1/z]$,

where A and A'' agree on the free variables of e_1, where A' and A'' agree on the free variables of e_2 except z, and where $A'(z) = \tau$.

Proof. By induction on the structure of the derivation of $A' \vdash \widehat{w_2} : \tau' \triangleright w_2$. □

Theorem 5.3 (Subject Reduction) *If $A \vdash \widehat{w} : \tau \triangleright w$ and $w \longrightarrow w'$, then $A \vdash \widehat{w'} : \tau \triangleright w'$.*

Proof. By induction on the structure of the derivation of $A \vdash \widehat{w} : \tau \triangleright w$, using Lemma 5.2. □

Next we prove that if a closed 2-level λ-term of type d is reduced to normal form, then all the components of that normal form will be dynamic.

Theorem 5.4 (Dynamic Normal Form) *Suppose w is a 2-level λ-term in normal form, and suppose A is an environment such that $A(x) = d$ for all x in the domain of A. If $A \vdash \widehat{w} : d \triangleright w$, then all components of w are dynamic.*

Proof. By induction on the structure of the derivation of $A \vdash \widehat{w} : d \triangleright w$. □

Finally, we prove that typability ensures that no "confusion" between static and dynamic will occur, for example as in $(\underline{\lambda}x.e)\overline{@}e_1$.

Theorem 5.5 (No Confusion) *If $A \vdash \widehat{w} : \tau \triangleright w$, then the following "confused" terms do not occur in w.*

$$(\underline{\lambda}x.e)\overline{@}e_1$$
$$(\overline{\lambda}x.e)\underline{@}e_1$$
$$\overline{\mathsf{fst}}\ \underline{\mathsf{pair}}(e_1, e_2)$$
$$\underline{\mathsf{fst}}\ \overline{\mathsf{pair}}(e_1, e_2)$$
$$\overline{\mathsf{snd}}\ \underline{\mathsf{pair}}(e_1, e_2)$$
$$\underline{\mathsf{snd}}\ \overline{\mathsf{pair}}(e_1, e_2)$$
$$\overline{\mathsf{case}}\ \underline{\mathsf{inleft}}(e)\ \overline{\mathsf{of}}\ \overline{\mathsf{inleft}}(x_1) \Rightarrow e_1 \mid \overline{\mathsf{inright}}(x_2) \Rightarrow e_2\ \overline{\mathsf{end}}$$
$$\underline{\mathsf{case}}\ \overline{\mathsf{inleft}}(e)\ \underline{\mathsf{of}}\ \underline{\mathsf{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\mathsf{inright}}(x_2) \Rightarrow e_2\ \underline{\mathsf{end}}$$
$$\overline{\mathsf{case}}\ \underline{\mathsf{inright}}(e)\ \overline{\mathsf{of}}\ \overline{\mathsf{inleft}}(x_1) \Rightarrow e_1 \mid \overline{\mathsf{inright}}(x_2) \Rightarrow e_2\ \overline{\mathsf{end}}$$
$$\underline{\mathsf{case}}\ \overline{\mathsf{inright}}(e)\ \underline{\mathsf{of}}\ \underline{\mathsf{inleft}}(x_1) \Rightarrow e_1 \mid \underline{\mathsf{inright}}(x_2) \Rightarrow e_2\ \underline{\mathsf{end}}$$

Proof. Immediate. □

Together, Theorems 5.1, 5.3, 5.4, and 5.5 guarantee that if we have derived $A \vdash e : d \triangleright w$, then we can start specialization of w and know that

- if a normal form is reached, then all its components will be dynamic; and
- no confused terms will occur at any point.

We have thus established the correctness of a partial evaluator which automatically does The Trick. Notice that the correctness statement also holds without eta-expansion, *i.e.*, for the partial evaluator specified in Section 2.

6 Assessment

The two new eta-expansion rules of Figure 8 unify and generalize our earlier treatment of eta-expansion [10], and they are a key part of our explanation of The Trick. Intuitively, the two rules make it possible (1) to coerce the binding-time type d to any type τ, and (2) to coerce any type τ to the type d. There is no direct rule, however, for coercing for example $d \to d$ to $d \to (d \to d)$. Such a rule seems to be definable using some notion of subtyping.

Our rules for eta-expansion remind of rules for inserting coercions in type systems with subtyping. The purpose of our rules, however, is not to change the type of a term to a supertype; two of our coercions can change the type of a term to any other type.

In Jones, Gomard, and Sestoft's textbook [16], using The Trick requires the partial-evaluation user to collect static information under dynamic control (either by hand or by program analysis) and to rewrite the source program to exploit it. We represent this statically collected information as a disjoint sum.

Jones, Gomard, and Sestoft also restrict static values occurring in dynamic contexts to be of base type. Values are higher type are dynamized, thereby making their type a base type, namely dynamic. In contrast, the binding-time analysis of Section 4 provides a syntactic representation of binding-time coercions at higher type.

Finally, in contrast to our monovariant binding-time analysis, a polyvariant binding-time analysis associates *several* binding-time descriptions with each program point. Polyvariance obviates binding-time coercions, by generating several variants instead of coercing them into a single one. Experience, however, shows that polyvariance is expensive [1]. Moreover, our personal experience with Consel's partial evaluator Schism [6] shows that eta-expansion can speed up a polyvariant binding-time analysis by reducing the number of variants.

7 Conclusion

We have specified and proven the correctness of a partial evaluator for a λ-calculus with products and disjoint sums. The specializer moves static contexts across dynamic case expressions and the binding-time analysis accounts for this move (Section 2). We have demonstrated that in such a partial evaluator, eta-expansion for disjoint-sum values achieves The Trick

(Section 3). Our binding-time analysis automatically inserts binding-time coercions as eta-redexes (Section 4), and thus our partial evaluator both automates and unifies the binding-time improvements listed in Jones, Gomard, and Sestoft's textbook [16, Chapter 12]. Future work includes finding an efficient algorithm for our new binding-time analysis.

Acknowledgements

The first author is supported by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils. The second author was hosted by the BRICS Centre (Basic Research In Computer Science) of the Danish National Research Foundation during summer 1995. The third author is supported by the Danish Natural Science Research Council and BRICS.

The diagram of Section 2 was drawn with Kristoffer Rose's Xy-pic package.

References

[1] J. Michael Ashley and Charles Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Transactions on Programming Languages and Systems*, 16(5):1431–1448, 1994.

[2] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[3] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[4] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991. Special issue on ESOP'90, the Third European Symposium on Programming, Copenhagen, May 15-18, 1990.

[5] Anders Bondorf. Improving binding times without explicit cps-conversion. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 1–10, San Francisco, California, June 1992. ACM Press.

[6] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[7] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [13], pages 496–519.

[8] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[9] Olivier Danvy. Type-directed partial evaluation (extended abstract). Technical report DAIMI PB-494, Computer Science Department, Aarhus University, Aarhus, Denmark, July 1995.

[10] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 8(3):209–227, 1995. An earlier version appeared in the proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

[11] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[12] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[13] John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts, August 1991.

[14] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[15] Neil D. Jones, editor. *Special issue on Partial Evaluation*, Journal of Functional Programming, Vol. 3, Part 3. Cambridge University Press, July 1993.

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[17] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report CS-95-178, Computer Science Department, Brandeis University, Waltham, Massachusetts, January 1995. An earlier version appeared in the proceedings of the 1994 ACM Conference on Lisp and Functional Programming.

[18] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[19] Jens Palsberg. Correctness of binding-time analysis. In Jones [15], pages 347–363.

[20] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.

[21] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

[22] Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.

[23] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[24] Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones [15], pages 365–387.

[25] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In Hughes [13], pages 165–191.

Recent Publications in the BRICS Report Series

RS-95-41 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. *Eta-Expansion Does The Trick*. August 1995. 23 pp.

RS-95-40 Anna Ingólfsdóttir and Andrea Schalk. *A Fully Abstract Denotational Model for Observational Congruence*. August 1995. 29 pp.

RS-95-39 Allan Cheng. *Petri Nets, Traces, and Local Model Checking*. July 1995. 32 pp. Full version of paper appearing in Proceedings of AMAST '95, LNCS 936, 1995.

RS-95-38 Mayer Goldberg. *Gödelisation in the λ-Calculus*. July 1995. 7 pp.

RS-95-37 Sten Agerholm and Mike Gordon. *Experiments with ZF Set Theory in HOL and Isabelle*. July 1995. 14 pp. To appear in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS, 1995.

RS-95-36 Sten Agerholm. *Non-primitive Recursive Function Definitions*. July 1995. 15 pp. To appear in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS, 1995.

RS-95-35 Mayer Goldberg. *Constructing Fixed-Point Combinators Using Application Survival*. June 1995. 14 pp.

RS-95-34 Jens Palsberg. *Type Inference with Selftype*. June 1995. 22 pp.

RS-95-33 Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. *Type Inference with Non-structural Subtyping*. June 1995. 22 pp.

RS-95-32 Jens Palsberg. *Efficient Inference of Object Types*. June 1995. 32 pp. To appear in *Information and Computation*. Preliminary version appears in *Ninth Annual IEEE Symposium on Logic in Computer Science*, LICS '94 Proceedings, pages 186–195.

RS-95-31 Jens Palsberg and Peter Ørbæk. *Trust in the λ-calculus*. June 1995. 32 pp. To appear in *Static Analysis: 2nd International Symposium*, SAS '95 Proceedings, 1995.