BRICS RS-96-17 Danvy et al.: Eta-Expansion Does The Trick (Revised Version)

**BRICS**

**Basic Research in Computer Science**

# Eta-Expansion Does The Trick

## (Revised Version)

**Olivier Danvy**
**Karoline Malmkjær**
**Jens Palsberg**

**BRICS Report Series** **RS-96-17**

**ISSN 0909-0878** **May 1996**

**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@brics.dk**

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

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

# Eta-Expansion Does The Trick *

Olivier Danvy Karoline Malmkjær Jens Palsberg
**BRICS** † MIT§
Aarhus University‡

May 1996

**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": nonstandard 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 nonstandard use of continuation-passing style encountered in partial evaluation. Eta-expansion thus 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 enables "The Trick".

In this article, we extend Gomard and Jones's partial evaluator for the $\lambda$-calculus, $\lambda$-Mix, with products and disjoint sums; we point out how eta-expansion for (finite) disjoint sums enables 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 $\lambda$-Mix.

**Keywords:** Partial evaluation, binding-time analysis, program specialization, binding-time improvement, eta-expansion, static reduction.

---

*To appear in the ACM Transactions on Programming Languages and Systems

†Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

‡Computer Science Department, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark; E-mail: {danvy,karoline}@brics.dk.

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

## 1 Introduction

Partial evaluation is a program-transformation technique for specializing programs [11, 24]. As such, it contributes to solving the tension between program generality (to ease portability and maintenance) and program specificity (to have them attuned to the situation at hand). Modern partial evaluators come in two flavors: online and offline.

### 1.1 Online partial evaluation

An online partial evaluator specializes programs in an interpretive way [35, 43]. For example, consider the treatment of conditional expressions. An online partial-evaluation function maps a source program and an environment to a disjoint sum: the result is either a static value or a residual expression.

$$
\begin{array}{rcl}
\mathcal{PE} & : & \mathrm{Exp} \rightarrow \mathrm{Env} \rightarrow \mathrm{Val} + \mathrm{Exp} \\
\mathcal{PE}[\![\texttt{(IF e1 e2 e3)}]\!]\,\rho & = & \mathrm{case}\ \mathcal{PE}[\![\texttt{e1}]\!]\,\rho\ \mathrm{of} \\
 & & \quad \mathrm{inVal}(v_1) \\
 & & \quad\ \Rightarrow \mathrm{if}\ v_1|_{\mathrm{Bool}} \\
 & & \qquad\quad \mathrm{then}\ \mathcal{PE}[\![\texttt{e2}]\!]\,\rho \\
 & & \qquad\quad \mathrm{else}\ \mathcal{PE}[\![\texttt{e3}]\!]\,\rho \\
 & & [\!]\ \mathrm{inExp}(e_1) \\
 & & \quad\ \Rightarrow \mathrm{inExp}(\mathrm{in}\texttt{IF}(e_1, \\
 & & \qquad\qquad\qquad \mathrm{case}\ \mathcal{PE}[\![\texttt{e2}]\!]\,\rho\ \mathrm{of} \\
 & & \qquad\qquad\qquad\quad \mathrm{inVal}(v_2) \Rightarrow \mathrm{residualize}(v_2) \\
 & & \qquad\qquad\qquad [\!]\ \mathrm{inExp}(e_2) \Rightarrow e_2 \\
 & & \qquad\qquad\qquad \mathrm{end}, \\
 & & \qquad\qquad\qquad \mathrm{case}\ \mathcal{PE}[\![\texttt{e3}]\!]\,\rho\ \mathrm{of} \\
 & & \qquad\qquad\qquad\quad \mathrm{inVal}(v_3) \Rightarrow \mathrm{residualize}(v_3) \\
 & & \qquad\qquad\qquad [\!]\ \mathrm{inExp}(e_3) \Rightarrow e_3 \\
 & & \qquad\qquad\qquad \mathrm{end})) \\
 & & \mathrm{end}
\end{array}
$$

At every step, the partial evaluator must perform a binding-time test, *i.e.*, it must check whether each intermediate result is a static value or a residual expression. In the case of a conditional expression, the test part is partially evaluated first.

- If its result is a static value, and assuming this value is boolean, we test it and select the corresponding branch accordingly.

- If its result is a residual expression, we need to reconstruct the conditional expression, deferring the test and the corresponding branch selection until run time. To this end, both conditional branches are (speculatively) processed. Again, the binding time of their result is tested. If either result is a static value, it is residualized, *i.e.*, turned into a residual expression that will evaluate to this value at run time. (In Lisp, residualizing a static value amounts to quoting it.) If either result is a residual expression, it just fits in the residual conditional expression.

### 1.2 Offline partial evaluation

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 [22, 30, 31, 42].

Considering again conditional expressions as above, the net effect of binding-time analysis is to factor out the binding-time checks. The static values are classified as static, and the residual expressions are classified as dynamic. As a rule, binding-time analyses lean toward safety in the sense that in case of doubt a dynamic classification is safer than a static one.

### 1.3 This article

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 [11, 24]. 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 [24, Chapter 12] documents three such "binding-time improvements": continuation-passing style, eta-expansion, and "The Trick".

### 1.4 Continuation-passing style

Evaluating some expressions reduces to evaluating some of their subexpressions; 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

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

and

$$10 + (\mathrm{case}\ D\ \mathrm{of}\ \mathrm{inleft}(t_1) \Rightarrow 1 \mathbin{[\!]} \mathrm{inright}(t_2) \Rightarrow 2\ \mathrm{end})$$

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 unsound, since it would prevent the computational effect from occurring at run time.) Both the second arguments of + are therefore dynamic, and thus both occurrences of + are 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 of these continuation-based methods are documented in the literature [5, 10, 24, 27]. Note that this change in the specializer requires a corresponding change in the binding-time analysis.

### 1.5 Eta-expansion

Jones, Gomard, and Sestoft list eta-expansion as an effective binding-time improvement [24]. In an earlier work [14], 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 article [14], 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 consider disjoint sums.

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

### 1.6 "The Trick"

In their partial-evaluation textbook [24], Jones, Gomard, and Sestoft document a folklore binding-time improvement, referring to it as "The Trick". Until now, 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 [24, Section 12.2]. The Trick is usually carried out explicitly by the programmer (see the while loop in Jones and Gomard's Imperative Mix [24, Section 4.8.3]).

This enumeration of static values could also be obtained by program analysis, for example using Heintze's set-based analysis [18]. Exploiting the results of such a program analysis would make it possible to automate 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 [38] (also known as closure analysis [37]) determines a conservative approximation of which $\lambda$-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 $\lambda$-abstractions and performing a first-order call to the corresponding $\lambda$-abstraction in each branch. This defunctionalization technique was proposed by Reynolds in the early seventies [33], and recently cast in a typed setting [29]. 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 [24, 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.

### 1.7 Overview

In this article 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 Gomard and Jones's $\lambda$-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.

### 1.8 Notation

Consistently with Nielson and Nielson [30], 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 $\lambda$-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 $e$ of type $\tau_1 \to \tau_2$ yields the expression

$$\lambda v.e@v$$

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

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

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

$$\text{case } e \text{ of inleft}(x_1) \Rightarrow \text{inleft}(x_1) \mathbin{[\!]} \text{inright}(x_2) \Rightarrow \text{inright}(x_2) \text{ end.}$$

## 2 An Extension of $\lambda$-Mix handling Products and Disjoint Sums

Our starting point is Gomard and Jones's partial evaluator $\lambda$-Mix, an offline partial evaluator for the $\lambda$-calculus [16, 17, 24]. 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. Also like Gomard and Jones, only static terms are typed.

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]. The technique is also used to specify "one-pass" CPS transformations [13]. Like the CPS transformation, the specification can be specified both purely functionally or in a more "direct" style, using control operators [26].

Figure 1 displays the syntax of a $\lambda$-calculus with products and disjoint sums. Figure 2 displays the syntax of a two-level $\lambda$-calculus where each construct, except variables, has two forms: overlined (static) and underlined (dynamic). A two-level $\lambda$-term is said to be *completely dynamic* if all the constructs in it are underlined. A binding-time analysis (Section 2.1) maps a $\lambda$-term into a two-level $\lambda$-term. Program specialization (Section 2.2) reduces all the static parts of a two-level $\lambda$-term and yields a completely dynamic $\lambda$-term. Erasing its annotations yields the residual, specialized $\lambda$-term. This is summarized in the following diagram:

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

Figure 1: BNF of the $\lambda$-calculus

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

Figure 2: BNF of the two-level $\lambda$-calculus

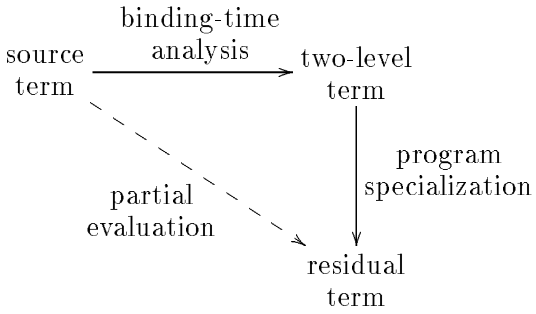
$$
A \vdash x : A(x) \triangleright x
$$

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

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

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

Figure 3: Gomard's binding-time analysis for the pure $\lambda$-calculus




## 2.1 Binding-time analysis

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

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

should be read as follows: under hypothesis $A$, the $\lambda$-term $e$ can be assigned the type $t$ with the annotated term $w$. The whole program must be assigned the type $d$, whereas parts of the program can have other types. This requirement ensures that program specialization can produce a completely dynamic $\lambda$-term. Notice that a $\lambda$-term can have several types and several annotated versions. For example, both

$$\emptyset \vdash \lambda x.x : d \triangleright \underline{\lambda} x.x$$

and

$$\emptyset \vdash \lambda x.x : d \rightarrow d \triangleright \overline{\lambda} x.x$$

are derivable. Notice also that each $\tau$ in Figure 3 can be either $d$ or some other type.

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.*

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

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

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

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

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

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

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

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

$$\frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{inleft}(e) : d \triangleright \underline{\mathrm{inleft}}(w)} \qquad \frac{A \vdash e : d \triangleright w}{A \vdash \mathrm{inright}(e) : d \triangleright \underline{\mathrm{inright}}(w)}$$

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:

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

Static projections:

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

Figure 6: Operational semantics of the two-level $\lambda$-calculus — evaluation rules

Static applications:

$$\begin{array}{l} (\underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}})\overline{@}e \longrightarrow \\ \underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1\overline{@}e\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\overline{@}e\ \underline{\mathrm{end}} \end{array}$$

Static decompositions:

$$\begin{array}{l} \overline{\mathrm{fst}}\ (\underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}}) \longrightarrow \\ \underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow \overline{\mathrm{fst}}\ e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow \overline{\mathrm{fst}}\ e_2\ \underline{\mathrm{end}} \end{array}$$

$$\begin{array}{l} \overline{\mathrm{snd}}\ (\underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}}) \longrightarrow \\ \underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow \overline{\mathrm{snd}}\ e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow \overline{\mathrm{snd}}\ e_2\ \underline{\mathrm{end}} \end{array}$$

Static projections:

$$\begin{array}{l} \overline{\mathrm{case}}\ (\underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}})\ \overline{\mathrm{of}} \\ \quad \overline{\mathrm{inleft}}(x_1') \Rightarrow e_1' \\ [\!]\ \overline{\mathrm{inright}}(x_2') \Rightarrow e_2' \\ \overline{\mathrm{end}} \\ \longrightarrow \underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}} \\ \qquad \underline{\mathrm{inleft}}(x_1) \Rightarrow \overline{\mathrm{case}}\ e_1\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(x_1') \Rightarrow e_1'\ [\!]\ \overline{\mathrm{inright}}(x_2') \Rightarrow e_2'\ \overline{\mathrm{end}} \\ \quad\ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow \overline{\mathrm{case}}\ e_2\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(x_1') \Rightarrow e_1'\ [\!]\ \overline{\mathrm{inright}}(x_2') \Rightarrow e_2'\ \overline{\mathrm{end}} \\ \quad\ \underline{\mathrm{end}} \end{array}$$

Figure 7: Operational semantics of the two-level $\lambda$-calculus — code-motion rules

### 2.2 Program specialization

Program specialization reduces the static parts of a two-level $\lambda$-term. Our specification of program specialization has the form of an operational semantics. If $e$ and $e'$ are two-level $\lambda$-terms, then $e \longrightarrow e'$ means that $e$ reduces to $e'$. Figure 6 displays the three basic evaluation rules, and Figure 7 displays four "code-motion" rules. We say that a two-level $\lambda$-term is in normal form if it cannot be reduced. 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 the rule for binding-time analysis of case expressions of Figure 5. Notice that there is no rule of the form

$$\begin{array}{l} e\overline{@}(\underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}}) \longrightarrow \\ \underline{\mathrm{case}}\ e_0\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e\overline{@}e_1 \ [\!]\ \underline{\mathrm{inright}}(x_2) \Rightarrow e\overline{@}e_2\ \underline{\mathrm{end}}. \end{array}$$

This is because such a rule cannot create redexes unless the left-hand side is already a redex itself.

The code motion rules in Figure 7 occur variously in logic, proof theory, CPS transformation, deforestation, and partial evaluation. Similarly to Paulin-Mohring and Werner [32, Section 4.5.4], we use them to move static values toward static contexts, in the simply typed two-level $\lambda$-calculus. For each context $E[\cdot]$, if $e \longrightarrow e'$, then $E[e] \longrightarrow E[e']$.

Our extension of $\lambda$-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 [14], for functions, eta-expansion is useful in two cases. The first is where a dynamic context $E[\cdot]$, expecting a higher-order value of type $d$ (one could be tempted to write "of type $\tau_1 \underline{\rightarrow} \tau_2$" to clarify that this is a function, but in the present treatment, dynamic terms are not typed), 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 write $\tau_1 \underline{\to} \tau_2$) can be coerced into a static value

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

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

### 3.1.1 A concrete example: Church numerals

Church numerals [2] are defined with a $\lambda$-representation for the number zero and with a $\lambda$-representation for the successor function:

$$\begin{array}{rcl} \text{zero} & = & \lambda s.\lambda z.z \\ \text{succ} & = & \lambda n.\lambda s.\lambda z.s@(n@s@z) \end{array}$$

Suppose we want to specialize succ with respect to a given numeral, say the one corresponding to 2, *i.e.*, succ@(succ@zero). A standard binding-time analysis does not allow source arguments to be higher order [24]. Our binding-time analysis, however, will produce the following two-level, eta-expanded term (the eta-redex is boxed):

$$(\overline{\lambda} n.\underline{\lambda} s.\underline{\lambda} z.s\underline{@}(n\overline{@}\boxed{(\overline{\lambda} v.s\underline{@}v)}\overline{@}z))\overline{@}\overline{\lambda} s.\overline{\lambda} z.s\overline{@}(s\overline{@}z)$$

The following Scheme session illustrates Church numerals and their residualization.

```
> (define zero (lambda (s) (lambda (z) z)))
> (define succ
    (lambda (n) (lambda (s) (lambda (z) (s ((n s) z))))))
> (define succ-gen
    (lambda (n)
      (let* ([s (gensym! "s")]
             [z (gensym! "z")])
        `(lambda (,s)
           (lambda (,z)
             (,s ,((n (lambda (v) `(,s ,v))) z)))))))
> (succ-gen (succ (succ zero)))
(lambda (s0) (lambda (z1) (s0 (s0 (s0 z1)))))
> (((lambda (s0) (lambda (z1) (s0 (s0 (s0 z1))))) 1+) 0)
3
>
```

Procedure `succ-gen` is the generating extension of Procedure `succ`, *i.e.*, its associated program specializer [24]. Applying `succ-gen` to the static data gives the same result as specializing `succ` with respect to the static data. In the definition of `succ-gen`, binding-time information is encoded with Scheme's quasi-quote (backquote) and unquote (comma) [7].

### 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

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

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

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

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

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

$$\mathrm{fst}\ e$$

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

$$\underline{\mathrm{fst}}\, e$$

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

$$\underline{\mathrm{fst}}\,(\underline{\mathrm{pair}}(\underline{\lambda} x.(\overline{\mathrm{fst}}\ e)\overline{@}x, \overline{\mathrm{snd}}\ e)).$$

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 by-product, dynamizes this context. Again, $e$ could be eta-expanded to read

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

This term has type $d \times d$, which avoids dynamizing the context and thus makes it possible to keep $e'$ a static pair, thanks to the coercion. (Note that the alternative of eta-expanding $e'$ into $\underline{\mathrm{pair}}(\overline{\mathrm{fst}}\ e', \overline{\mathrm{snd}}\ e')$ would not be a binding-time improvement, since it would dynamize the present context.)

### 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.

$$(\lambda v.f@(\mathrm{case}\; v\; \mathrm{of}\; \mathrm{inleft}(a) \Rightarrow a + 20 \;[\!]\; \mathrm{inright}(b) \Rightarrow \ldots\, \mathrm{end})@v)@\mathrm{inleft}(10)$$

Assume this $\beta$-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.

$$(\overline{\lambda} v.f\underline{@}(\underline{\mathrm{case}}\; v\; \underline{\mathrm{of}}\; \underline{\mathrm{inleft}}(a) \Rightarrow a \underline{+} 20 \;[\!]\; \underline{\mathrm{inright}}(b) \Rightarrow \ldots\, \underline{\mathrm{end}})\underline{@}v)\overline{@}\underline{\mathrm{inleft}}(10)$$

In this term, both $f$ and $v$ have type $d$.

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

$$f@(\mathrm{case}\; \mathrm{inleft}(10)\; \mathrm{of}\; \mathrm{inleft}(a) \Rightarrow a + 20 \;[\!]\; \mathrm{inright}(b) \Rightarrow \ldots\, \mathrm{end})@\mathrm{inleft}(10)$$

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

Note that 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).

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

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

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

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

Specialization yields the residual term

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

which is more reduced than the residual term (a) above.

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.

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

Assume this $\beta$-redex will be reduced. Notice that $f$ occurs twice: it is applied both to a static value and to a dynamic value. The binding-time analysis of Figures 3, 4, and 5 thus approximates its argument to be dynamic and yields the following two-level term.

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

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

Specialization yields the following residual term (call it (b)).

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

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

$$
\begin{array}{l}
\lambda f.\ \ldots \\
\quad f@\boxed{\mathrm{case}\ d\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow \mathrm{inleft}(a)\ [\!]\ \mathrm{inright}(b) \Rightarrow \mathrm{inright}(b)\ \mathrm{end}} \\
\quad \ldots \\
\quad f@\mathrm{inleft}(\lambda x.x) \\
\quad \ldots \\
@ \\
\lambda v.\mathrm{case}\ v\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow a@10\ [\!]\ \mathrm{inright}(b) \Rightarrow \ldots\ \mathrm{end}
\end{array}
$$

This eta-expansion enables 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 does The Trick, by plugging 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.

$$
\begin{array}{l}
\overline{\lambda} f.\ \ldots \\
\quad f\overline{@}\underline{\mathrm{case}}\ d\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(a) \Rightarrow \overline{\mathrm{inleft}}(a)\ [\!]\ \underline{\mathrm{inright}}(b) \Rightarrow \overline{\mathrm{inright}}(b)\ \underline{\mathrm{end}} \\
\quad \ldots \\
\quad f\overline{@}\,\overline{\mathrm{inleft}}(\underline{\lambda} x.x) \\
\quad \ldots \\
\overline{@} \\
\overline{\lambda} v.\overline{\mathrm{case}}\ v\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(a) \Rightarrow a\underline{@}10\ [\!]\ \overline{\mathrm{inright}}(b) \Rightarrow \ldots\ \overline{\mathrm{end}}
\end{array}
$$

In this term, $f$ has type $(d + t) \to d$, for some $t$.

Specialization moves the context of the dynamic case expression in each of its branches and produces the following residual term (call it (c)).

$$
\begin{array}{l}
\ldots \\
\mathrm{case}\ d\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow a@10\ [\!]\ \mathrm{inright}(b) \Rightarrow \ldots\ \mathrm{end} \\
\ldots \\
(\lambda x.x)@10 \\
\ldots
\end{array}
$$

This residual term (c) is more reduced than the residual term (b) above.

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

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

$$
\begin{array}{l}
\lambda f.\ \ldots \\
\quad f@\mathrm{case}\ d\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow \mathrm{inleft}(\boxed{\lambda z.a@z})\ [\!]\ \mathrm{inright}(b) \Rightarrow \mathrm{inright}(b)\ \mathrm{end} \\
\quad \ldots \\
\quad f@\mathrm{inleft}(\lambda x.x) \\
\quad \ldots \\
@ \\
\lambda v.\mathrm{case}\ v\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow a@10\ [\!]\ \mathrm{inright}(b) \Rightarrow \ldots\ \mathrm{end}
\end{array}
$$

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

$$
\begin{array}{l}
\overline{\lambda} f.\ \ldots \\
\quad f\overline{@}\underline{\mathrm{case}}\ d\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(a) \Rightarrow \overline{\mathrm{inleft}}(\overline{\lambda} z.a\underline{@}z)\ [\!]\ \underline{\mathrm{inright}}(b) \Rightarrow \overline{\mathrm{inright}}(b)\ \underline{\mathrm{end}} \\
\quad \ldots \\
\quad f\overline{@\mathrm{inleft}}(\overline{\lambda} x.x) \\
\quad \ldots \\
\overline{@} \\
\overline{\lambda} v.\overline{\mathrm{case}}\ v\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(a) \Rightarrow a\overline{@}10\ [\!]\ \overline{\mathrm{inright}}(b) \Rightarrow \ldots\ \overline{\mathrm{end}}
\end{array}
$$

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

Specialization yields the following residual term.

$$
\begin{array}{l}
\ldots \\
(\mathrm{case}\ d\ \mathrm{of}\ \mathrm{inleft}(a) \Rightarrow a@10\ [\!]\ \mathrm{inright}(b) \Rightarrow \ldots\ \mathrm{end}) \\
\ldots \\
10 \\
\ldots
\end{array}
$$

This residual term is more reduced than the term (c) above.

### 3.3.3 A concrete example: Mix's pending list

The Trick was first used to program Mix, the first self-applicable partial evaluator [25]. Mix's program specializer is polyvariant and operates on a "pending list", which is a list of specialization points, subindexed with static values. When Mix is self-applied, looking up in this list is a dynamic operation, even though the specialization points are static. The Trick is used to move the context of this lookup (*i.e.*, the specializer) inside the list to specialize the specialization points at self-application time.

Holst and Hughes have characterized this use of The Trick as the application of one of Wadler's theorems for free: Reynolds's Abstraction theorem in the first-order case [20, 34, 41]. The composition of specialization and list lookup is replaced by the composition of lookup and map of specialization over the list. This achieves a binding-time improvement because it enables the specialization of specialization points at self-application time.

In the context of this article, and since the source program has a fixed number of specialization points, the pending list has a fixed length, and thus it can be formalized as a finite disjoint sum. Eta-expansion over this disjoint sum enables The Trick, through which specialization points are specialized at self-application time.

### 3.4 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 enables 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 does The Trick and plugs these values into the surrounding context (see Figure 7).

## 4 Binding-Time Analysis with Eta-Expansion

In our earlier work [14], 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

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

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

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

$$d \vdash e \Rightarrow e$$

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

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

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

Figure 9: Type-directed eta expansion

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 $\tau$ and (2) to coerce any type $\tau$ to the type $d$. The combination of the two rules allows us to coerce the type of any $\lambda$-term to any other type.

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 $\lambda$-terms [3, 12].

Consider the first rule in Figure 8. Intuitively, it works as follows. We are given a $\lambda$-term $e$ that we would like to assign the type $\tau$. In case we can only assign it type $d$ and $\tau \neq d$, we can use the rule to coerce the type to be $\tau$. The first hypothesis of the rule is that $e$ has type $d$ and annotated term $w$. The second hypothesis of the rule takes a fresh variable $z$ and eta-expands it according to the type $\tau$. This creates a $\lambda$-term $m$ with type $\tau$. Notice that $z$ is the only free variable in $m$. The third hypothesis of the rule annotates $m$

under the assumption that $z$ has type $d$. The result is an annotated term $w'$ with the type $\tau$ and with a hole of type $d$ (the free variable $z$) where we can insert the previously constructed $w$. Thus, $w'$ makes the coercion happen. The second rule in Figure 8 works in a similar way.

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.

For example, consider again the first example in Section 3.2, that is, the expression fst $e$. We assume that the judgment

$$\emptyset \vdash e : (d \rightarrow d) \times d \triangleright w$$

is derivable, *i.e.*, $e$ has type $(d \rightarrow d) \times d$ with annotated term $w$. Moreover, we assume that the expression fst $e$ occurs in a dynamic context, so we need to assign it type $d$. The following derivation does that, giving the expected annotated and eta-expanded version of $e$.

$$\frac{\emptyset \vdash e : d \triangleright \underline{\mathrm{pair}}(\underline{\lambda} x.(\overline{\mathrm{fst}}\ w)\overline{@}x, \overline{\mathrm{snd}}\ w)}{\emptyset \vdash \mathrm{fst}\ e : d \triangleright \underline{\mathrm{fst}}\,\underline{\mathrm{pair}}(\underline{\lambda} x.(\overline{\mathrm{fst}}\ w)\overline{@}x, \overline{\mathrm{snd}}\ w)}$$

To derive the hypothesis we use the second rule in Figure 8. We need the following three judgments:

$$\emptyset \vdash e : (d \rightarrow d) \times d \triangleright w$$
$$(d \rightarrow d) \times d \vdash z \Rightarrow \mathrm{pair}(\lambda x.(\mathrm{fst}\ z)@x, \mathrm{snd}\ z)$$
$$\emptyset[z \mapsto (d \rightarrow d) \times d] \vdash \mathrm{pair}(\lambda x.(\mathrm{fst}\ z)@x, \mathrm{snd}\ z) : d \triangleright \underline{\mathrm{pair}}(\underline{\lambda} x.(\overline{\mathrm{fst}}\ z)\overline{@}x, \overline{\mathrm{snd}}\ z)$$

The first judgment is given by assumption; the derivation of the other two are left to the reader.

## 5 Correctness

We now state and prove that our binding-time analysis is correct with respect to the operational semantics of two-level $\lambda$-terms. The statement of correctness is taken from Palsberg [31] and Wand [42], who proved correctness of two other binding-time analyses. The proof techniques are well known; we omit the details.

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

We first prove a basic property of the operational semantics of two-level $\lambda$-terms. Let $\longrightarrow\!\!\!\!\!\rightarrow$ be the reflexive and transitive closure of $\longrightarrow$.

**Theorem 5.1 (5.1 (Church-Rosser))** *If $e \longrightarrow\!\!\!\!\!\rightarrow e'$ and $e \longrightarrow\!\!\!\!\!\rightarrow e''$, then there exists $e'''$ such that $e' \longrightarrow\!\!\!\!\!\rightarrow e'''$ and $e'' \longrightarrow\!\!\!\!\!\rightarrow e'''$.*

*Proof.* By the method of Tait and Martin-Löf; the sequence of definitions and lemmas is standard [2, pp.59–62]. □

We then 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 5.2 (5.2 (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 5.2.1 (5.2.1 (Substitution))** *If*

$$A \vdash \widehat{w_1} : \tau \triangleright w_1 \quad \textit{and} \quad A' \vdash \widehat{w_2} : \tau' \triangleright w_2 \ ,$$

*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 $w_1$, where $A'$ and $A''$ agree on the free variables of $w_2$ 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 (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 5.2.1. □

Next we prove that if a closed two-level $\lambda$-term of type $d$ is reduced to normal form, then all the components of that normal form are dynamic.

**Theorem 5.4 (5.4 (Dynamic Normal Form))** *Suppose $w$ is a two-level $\lambda$-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 $w$ is completely dynamic.*

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

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

$$
\begin{array}{c}
(\underline{\lambda} x.e)\overline{@}e_1 \\
(\overline{\lambda} x.e)\underline{@}e_1 \\
\overline{\mathrm{fst}}\ \underline{\mathrm{pair}}(e_1, e_2) \\
\underline{\mathrm{fst}}\ \overline{\mathrm{pair}}(e_1, e_2) \\
\overline{\mathrm{snd}}\ \underline{\mathrm{pair}}(e_1, e_2) \\
\underline{\mathrm{snd}}\ \overline{\mathrm{pair}}(e_1, e_2) \\
\overline{\mathrm{case}}\ \underline{\mathrm{inleft}}(e)\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mathrel{[\!]} \overline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \overline{\mathrm{end}} \\
\underline{\mathrm{case}}\ \overline{\mathrm{inleft}}(e)\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mathrel{[\!]} \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}} \\
\overline{\mathrm{case}}\ \underline{\mathrm{inright}}(e)\ \overline{\mathrm{of}}\ \overline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mathrel{[\!]} \overline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \overline{\mathrm{end}} \\
\underline{\mathrm{case}}\ \overline{\mathrm{inright}}(e)\ \underline{\mathrm{of}}\ \underline{\mathrm{inleft}}(x_1) \Rightarrow e_1 \mathrel{[\!]} \underline{\mathrm{inright}}(x_2) \Rightarrow e_2\ \underline{\mathrm{end}}
\end{array}
$$

*Proof.* Immediate. $\square$

Together, Theorems 5.2, 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 and Related Work

The two new eta-expansion rules of Figure 8 unify and generalize our earlier treatment of eta-expansion [14], 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 $\tau$ and (2) to coerce any type $\tau$ 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 resemble rules for inserting coercions in type systems with subtyping [19]. 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.

We have not considered inferring binding-time annotations. This seems possible, using the technique of Dussart, Henglein, and Mossin [15] — a future work.

In Jones, Gomard, and Sestoft's textbook [24], 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 of 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. This syntactic representation can be interesting in its own right, in a setting where the binding time "dynamic" retains a type structure [12].

Polyvariant specializers usually select dynamic conditional expressions as specialization points [6, 24], thus disabling the code-motion rules of Figure 7. Experience, however, shows that not all dynamic conditional expressions need be treated as specialization points [28]. For these, the code-motion rules of Figure 7 can apply.

A polyvariant binding-time analysis, in contrast to our monovariant binding-time analysis, associates *several* binding-time descriptions with each program point [8]. 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 [9] shows that eta-expansion can speed up a polyvariant binding-time analysis by reducing the number of variants.

Finally, our results apply to online partial evaluation in that they provide guidelines to structure a typed online partial evaluator. Online partial evaluators usually keep multiple representations of static values, which obviates the need for residualization functions. They need, however, to be continuation-based to be able to achieve The Trick.

## 7 Conclusion

We have specified and proven the correctness of a partial evaluator for a $\lambda$-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, thus characterizing it as a typing property (Section 3). Our binding-time analysis automatically inserts binding-time coercions as eta-redexes (Section 4), and thus our partial evaluator both unifies and automates the binding-time improvements listed in Jones, Gomard, and Sestoft's textbook [24, Chapter 12]. Future work includes finding an efficient algorithm for our new binding-time analysis.


### Acknowledgements

Thanks to the referees for insightful comments.

The first author is supported by the BRICS Centre (Basic Research In Computer Science) of the Danish National Research Foundation and expresses grateful thanks to the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils, for support during 1995. The second author was hosted by BRICS 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. The Scheme session of Section 3.1.1 was obtained with R. Kent Dybvig's Chez Scheme system.


## 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 $\lambda$-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, Denmark, May 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] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[7] William Clinger and Jonathan Rees (editors). Revised$^4$ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[8] Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [36], pages 66–77.

[9] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Schmidt [36], pages 145–154.

[10] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [21], pages 496–519.

[11] 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.

[12] Olivier Danvy. Type-directed partial evaluation. In Steele Jr. [39], pages 242–257.

[13] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[14] 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.

[15] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 118–135, Glasgow, Scotland, September 1995. Springer-Verlag.

[16] 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, 1992.

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

[18] Nevin Heintze. *Set-Based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 1992.

[19] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1993. Special Issue on ESOP'92, the Fourth European Symposium on Programming, Rennes, February 1992.

[20] Carsten K. Holst and John Hughes. Towards binding-time improvement for free. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, pages 83–100, Glasgow, Scotland, 1990. Springer-Verlag.

[21] 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. Springer-Verlag.

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

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

[24] 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.

[25] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[26] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[27] 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.

[28] Karoline Malmkjær. Towards efficient partial evaluation. In Schmidt [36], pages 33–43.

[29] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In Steele Jr. [39], pages 271–283.

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

[31] Jens Palsberg. Correctness of binding-time analysis. In Jones [23], pages 347–363.

[32] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

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

[34] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Paris, France, 1983. IFIP.

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

[36] David A. Schmidt, editor. *Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[37] Peter Sestoft. Replacing function parameters by global variables. In Stoy [40], pages 39–53.

[38] 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.

[39] Guy L. Steele Jr., editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.

[40] Joseph E. Stoy, editor. *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, London, England, September 1989. ACM Press.

[41] Philip Wadler. Theorems for free! In Stoy [40], pages 347–359.

[42] Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones [23], pages 365–387.

[43] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In Hughes [21], pages 165–191.

## Recent Publications in the BRICS Report Series

**RS-96-17** **Olivier Danvy, Karoline Malmkjær, and Jens Palsberg.** ***Eta-Expansion Does The Trick (Revised Version)*. May 1996. 29 pp. To appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*.**

**RS-96-16** **Lisbeth Fajstrup and Martin Raußen. *Detecting Deadlocks in Concurrent Systems*. May 1996. 10 pp.**

**RS-96-15** **Olivier Danvy. *Pragmatic Aspects of Type-Directed Partial Evaluation*. May 1996. 27 pp.**

**RS-96-14** **Olivier Danvy and Karoline Malmkjær. *On the Idempotence of the CPS Transformation*. May 1996. 15 pp.**

**RS-96-13** **Olivier Danvy and René Vestergaard. *Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*. May 1996. 28 pp. To appear in *8th International Symposium on Programming Languages, Implementations, Logics, and Programs*, PLILP '96 Proceedings, LNCS, 1996.**

**RS-96-12** **Lars Arge, Darren E. Vengroff, and Jeffrey S. Vitter. *External-Memory Algorithms for Processing Line Segments in Geographic Information Systems*. May 1996. 34 pp. A shorter version of this paper appears in Spirakis, editor, *Algorithms - ESA '95: Third Annual European Symposium Proceedings*, LNCS 979, 1995, pages 295–310.**

**RS-96-11** **Devdatt Dubhashi, David A. Grable, and Alessandro Panconesi. *Near-Optimal, Distributed Edge Colouring via the Nibble Method*. May 1996. 17 pp. Appears in Spirakis, editor, *Algorithms - ESA '95: Third Annual European Symposium Proceedings*, LNCS 979, 1995, pages 448–459. Invited to be published in a special issue of *Theoretical Computer Science* devoted to the proceedings of ESA '95.**

**RS-96-10** **Torben Braüner and Valeria de Paiva. *Cut-Elimination for Full Intuitionistic Linear Logic*. April 1996. 27 pp. Also available as Technical Report 395, Computer Laboratory, University of Cambridge.**