Eta-Expansion Does The Trick

OLIVIER DANVY and KAROLINE MALMKJÆR
BRICS, Aarhus University

and

JENS PALSBERG
MIT

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' partial evaluator for the λ-calculus, λ-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 λ-Mix.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*functional constructs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*; I.1.3 [**Algebraic Manipulation**]: Languages and Systems—*evaluation strategies*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms; program transformation*

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.
Authors' addresses: O. Danvy and K. Malmkjær, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark; email: {danvy; karoline}@brics.dk; home pages: http://www.brics.dk/~{danvy; karoline}; J. Palsberg, Laboratory for Computer Science, Massachusetts Institute of Technology, NE43-340, 545 Technology Square, Cambridge, MA 02139; email: palsberg@theory.lcs.mit.edu.

General Terms: Algorithms, Languages, Performance
Additional Key Words and Phrases: Binding-time analysis and improvement, eta-expansion, partial evaluation, program specialization, static reduction

1. INTRODUCTION

Partial evaluation is a program-transformation technique for specializing programs [Consel and Danvy 1993; Jones et al. 1993]. 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 [Ruf 1993; Weise et al. 1991]. 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.

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

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 [Jones 1988; Nielson and Nielson 1992; Palsberg 1993; Wand 1993].

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 [Consel and Danvy 1993; Jones et al. 1993]. 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 [Jones et al. 1993, Ch. 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

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

and

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

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 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 [Bondorf 1992; Consel and Danvy 1991; Jones et al. 1993; Lawall and Danvy 1995]. 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 [Jones et al. 1993]. In an earlier work [Danvy et al. 1995], 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 work [Danvy et al. 1995], 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 [Jones et al. 1993], 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 [Jones et al. 1993, Sec. 12.2]. The Trick is usually carried out explicitly by the programmer (see the while loop in Jones and Gomard's Imperative Mix [Jones et al. 1993, Sec. 4.8.3]).

This enumeration of static values could also be obtained by program analysis, for example using Heintze's set-based analysis [Heintze 1992]. 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 [Shivers 1991] (also known as closure analysis [Sestoft 1989]) 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 [Reynolds 1972] and recently cast in a typed setting [Minamide et al. 1996]. Since the end of the eighties, it is used by such partial evaluators as Similix to handle higher-order programs [Bondorf 1991]. The conclusion of this is that Jones, Gomard, and Sestoft actually do use an automated version of The Trick [Jones et al. 1993, Sec. 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 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.

1.8 Notation

Consistently with Nielson and Nielson [1992], 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 [Barendregt 1984].

We assume Barendregt's Variable Convention [Barendregt 1984]: 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 of type $\tau_1 \to \tau_2$ yields the expression

$$\lambda v.e@v$$

where v does not occur free in e [Barendregt 1984]. By analogy, "eta-expanding" a product expression e of type $\tau_1 \times \tau_2$ yields the expression

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

(surjective pairing). And "eta-expanding" a disjoint-sum expression e of type $\tau_1+\tau_2$ yields the expression

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

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

Fig. 1. BNF of the λ-calculus.

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

Fig. 2. BNF of the two-level λ-calculus.

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 [Gomard 1992; Gomard and Jones 1991; Jones et al. 1993]. 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 [1992] has given a specification of this technique, but no proof of correctness. The technique is also used to specify "one-pass" CPS transformations [Danvy and Filinski 1990]. Like the CPS transformation, the specification can be specified both purely functionally or in a more "direct" style, using control operators [Lawall and Danvy 1994].

Figure 1 displays the syntax of a λ-calculus with products and disjoint sums. Figure 2 displays the syntax of a two-level λ-calculus where each construct, except variables, has two forms: overlined (static) and underlined (dynamic). A two-level λ-term is said to be *completely dynamic* if all the constructs in it are underlined. A binding-time analysis (Section 2.1) maps a λ-term into a two-level λ-term. 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. This is summarized in the diagram of Figure 3.

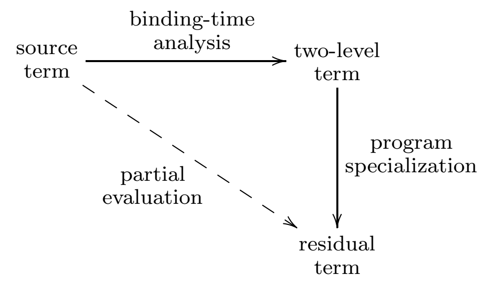

Fig. 3. Offline partial evaluation.

2.1 Binding-Time Analysis

Figure 4 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 judgment

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

should be read as follows: under hypothesis A, the λ-term e can be assigned the type t with the annotated term w. 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 λ-term. Notice that a λ-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 τ in Figure 4 can be either d or some other type.

Figures 5 and 6 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. Our 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 7 displays the three basic evaluation rules, and Figure 8 displays four "code-motion" rules. We say that a two-level λ-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

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

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

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

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

Fig. 4. Gomard's binding-time analysis for the pure λ-calculus.

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

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

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

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

Fig. 5. Extension of Gomard's binding-time analysis to products.

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

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

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

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

Fig. 6. 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:

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

Static projections:

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

Fig. 7. Operational semantics of the two-level λ-calculus — evaluation rules.

Static applications:

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

Static decompositions:

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

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

Static projections:

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

Fig. 8. Operational semantics of the two-level λ-calculus — code-motion rules.

Figure 6. Notice that there is no rule of the form

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

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 8 occur variously in logic, proof theory, CPS transformation, deforestation, and partial evaluation. Similarly to Paulin-Mohring and Werner [1993, Sec. 4.5.4], we use them to move static values toward static contexts, in the simply typed two-level λ-calculus. For each 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 [Danvy et al. 1995], 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 $\tau_1 \overline{\rightarrow} \tau_2$. The second useful case is where a dynamic higher-order value e of type d (again, one could be tempted to write $\tau_1 \underline{\rightarrow} \tau_2$) 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.

A Concrete Example: Church Numerals. Church numerals [Barendregt 1984] are defined with a λ-representation for the number zero and with a λ-representation for the successor function:

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

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 [Jones et al. 1993]. 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 [Jones et al. 1993]. 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 quasiquote (backquote) and unquote (comma) [Clinger and Rees 1991].

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 $(d \to d) \times d$, 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.

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 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.

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

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.

$$f@(\mathrm{case}\,\mathrm{inleft}(10)\,\mathrm{of}\,\mathrm{inleft}(a) \Rightarrow a + 20 \parallel \mathrm{inright}(b) \Rightarrow ...\,\mathrm{end})@\mathrm{inleft}(10)$$

The fact that inleft(10), a partially static value, occurs in the dynamic context $f@(\mathrm{case}\, v\, \mathrm{of}\, \mathrm{inleft}(a) \Rightarrow a + 20 \parallel \mathrm{inright}(b) \Rightarrow ...\,\mathrm{end})@[\cdot]$ "pollutes" its occurrence in the potentially static context $\mathrm{case}\,[\cdot]\,\mathrm{of}\,\mathrm{inleft}(a) \Rightarrow a + 20 \parallel \mathrm{inright}(b) \Rightarrow ...\,\mathrm{end}$, 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).

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

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

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

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.

Dynamic Injection in a Static Context. The following expression is partially evaluated in a context where d is dynamic.

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

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. The binding-time analysis of Figures 4, 5, and 6 thus approximates its argument to be dynamic and yields the following two-level term.

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

In this term, f has type d.

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

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

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

In this situation, a binding-time improvement is possible to make $\mathrm{inleft}(\lambda x.x)$ occur in a static context always. We can coerce the bothering occurrence of d by eta-expanding it (the eta-redex is boxed).

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

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 8).

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.

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

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)).

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

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.

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

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

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

Here, f has type $((d \to d) + t) \to d$, 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.

...
(case d of inleft$(a) \Rightarrow a$@10 ⫿ inright$(b) \Rightarrow$ ... end)
...
10
...

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

A Concrete Example: Mix's Pending List. The Trick was first used to program Mix, the first self-applicable partial evaluator [Jones et al. 1989]. 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 [Holst and Hughes 1990; Reynolds 1983; Wadler 1989]. 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 the Section on "Dynamic Injection in a Static Context" 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 8).

4. BINDING-TIME ANALYSIS WITH ETA-EXPANSION

In our earlier work [Danvy et al. 1995], 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 4, 5, and 6 with Figures 9 and 10.

Figure 9 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

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

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

Fig. 9. Extension of Gomard's binding-time analysis to binding-time coercions.

$$d \vdash e \Rightarrow e$$

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

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

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

Fig. 10. Type-directed eta-expansion.

and (2) to coerce any type τ to the type d. The combination of the two rules allows us to coerce the type of any λ-term to any other type.

Eta-expansion itself is defined in Figure 10. 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 [Berger and Schwichtenberg 1991; Danvy 1996].

Consider the first rule in Figure 9. Intuitively, it works as follows. We are given a λ-term e that we would like to assign the type τ. In case we can only assign it type d, and $\tau \neq d$, we can use the rule to coerce the type to be τ. 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 τ. This creates a λ-term m with type τ. 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 τ 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 9 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

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

is derivable, i.e., e has type $(d \to d) \times d$ 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 9. We need the following three judgments:

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

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 λ-terms. The statement of correctness is taken from Palsberg [1993] and Wand [1993], 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 λ-term, then $\widehat{w}$ denotes the underlying λ-term.

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

THEOREM 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 [Barendregt 1984, pp. 59–62]. $\square$

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 (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 $A \vdash e : \tau \triangleright w$. $\square$

We then prove subject reduction, using a substitution lemma.

LEMMA 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 $A' \vdash \widehat{w_2} : \tau' \triangleright w_2$. $\square$

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 5.2.1. □

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

Theorem 5.4 (Dynamic Normal Form). *Suppose w is a two-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 w is completely 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.*

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

Proof. Immediate. □

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 9 unify and generalize our earlier treatment of eta-expansion [Danvy et al. 1995], 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 resemble rules for inserting coercions in type systems with subtyping [Henglein 1993]. 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 [Dussart et al. 1995] — a future work.

In Jones, Gomard, and Sestoft's textbook [Jones et al. 1993], 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 [Danvy 1996].

Polyvariant specializers usually select dynamic conditional expressions as specialization points [Bondorf and Danvy 1991; Jones et al. 1993], thus disabling the code-motion rules of Figure 8. Experience, however, shows that not all dynamic conditional expressions need be treated as specialization points [Malmkjær 1993]. For these, the code-motion rules of Figure 8 can apply.

A polyvariant binding-time analysis, in contrast to our monovariant binding-time analysis, associates *several* binding-time descriptions with each program point [Consel 1993a]. Polyvariance obviates binding-time coercions, by generating several variants instead of coercing them into a single one. Experience, however, shows that polyvariance is expensive [Ashley and Consel 1994]. Moreover, our personal experience with Consel's partial evaluator Schism [Consel 1993b] 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 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, 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 [Jones et al. 1993, Ch. 12]. Future work includes finding an efficient algorithm for our new binding-time analysis.

ACKNOWLEDGMENTS

Thanks to the referees for insightful comments.

The diagram of Section 2 was drawn with Kristoffer Rose's XY-pic package. The Scheme session with Church numerals was obtained with R. Kent Dybvig's Chez Scheme system.

REFERENCES

Ashley, J. M. and Consel, C. 1994. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM Trans. Program. Lang. Syst. 16,* 5, 1431–1448.

Barendregt, H. 1984. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, Amsterdam, The Netherlands.

Berger, U. and Schwichtenberg, H. 1991. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, Los Alomitos, Calif., 203–211.

Bondorf, A. 1991. Automatic autoprojection of higher-order recursive equations. *Sci. Comput. Program. 17,* 1-3, 3–34. Special issue on ESOP'90, the Third European Symposium on Programming, May 1990.

Bondorf, A. 1992. Improving binding times without explicit cps-conversion. In Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, W. Clinger, Ed. *LISP Pointers 5,* 1 (June), 1–10.

Bondorf, A. and Danvy, O. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. *Sci. Comput. Program. 16*, 151–195.

Clinger, W. and Rees, J., Eds. 1991. Revised^4 report on the algorithmic language Scheme. *LISP Pointers 4,* 3 (July-Sept.), 1–55.

Consel, C. 1993a. Polyvariant binding-time analysis for applicative languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, D. A. Schmidt, Ed. ACM Press, New York, 66–77.

Consel, C. 1993b. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, D. A. Schmidt, Ed. ACM Press, New York, 145–154.

Consel, C. and Danvy, O. 1991. For a better support of static data flow. In *Proceedings of the 5th ACM Conference on Functional Programming and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 496–519.

Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, S. L. Graham, Ed. ACM Press, New York, 493–501.

Danvy, O. 1996. Type-directed partial evaluation. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, G. L. Steele, Ed. ACM Press, New York, 242–257.

Danvy, O. and Filinski, A. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, M. Wand, Ed. ACM Press, New York, 151–160.

Danvy, O., Malmkjær, K., and Palsberg, J. 1995. The essence of eta-expansion in partial evaluation. *LISP Symbol. Comput. 8,* 3, 209–227. An earlier version appeared in the Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

Dussart, D., Henglein, F., and Mossin, C. 1995. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Static Analysis*, A. Mycroft, Ed. Lecture Notes in Computer Science, vol. 983. Springer-Verlag, Glasgow, Scotland, 118–135.

Gomard, C. K. 1992. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Trans. Program. Lang. Syst. 14,* 2, 147–172.

Gomard, C. K. and Jones, N. D. 1991. A partial evaluator for the untyped lambda-calculus. *J. Funct. Program. 1,* 1, 21–69.

Heintze, N. 1992. Set-based program analysis. Ph.D. thesis, School of Computer Science, Carnegie Mellon Univ. Pittsburgh, Pa.

Henglein, F. 1993. Dynamic typing: Syntax and proof theory. *Sci. Comput. Program. 22,* 3, 197–230. Special Issue on ESOP'92, the Fourth European Symposium on Programming, February 1992.

Holst, C. K. and Hughes, J. 1990. Towards binding-time improvement for free. In *Functional Programming, Glasgow 1990*, S. L. Peyton Jones, G. Hutton, and C. K. Holst, Eds. Springer-Verlag, Berlin, 83–100.

Jones, N. D. 1988. Automatic program specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds. North-Holland, Amsterdam, The Netherlands, 225–282.

Jones, N. D., Gomard, C. K., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, N.J.

Jones, N. D., Sestoft, P., and Søndergaard, H. 1989. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP Symbol. Comput. 2,* 1, 9–50.

Lawall, J. L. and Danvy, O. 1994. Continuation-based partial evaluation. In Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, C. L. Talcott, Ed. *LISP Pointers 7,* 3 (June), 227–238.

Lawall, J. L. and Danvy, O. 1995. Continuation-based partial evaluation. Tech. Rep. CS-95-178, Computer Science Dept. , Brandeis Univ. Waltham, Mass. Jan. An earlier version appeared in the Proceedings of the 1994 ACM Conference on Lisp and Functional Programming.

Malmkjær, K. 1993. Towards efficient partial evaluation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, D. A. Schmidt, Ed. ACM Press, New York, 33–43.

Minamide, Y., Morrisett, G., and Harper, R. 1996. Typed closure conversion. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, G. L. Steele, Ed. ACM Press, New York, 271–283.

Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages.* Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press, Cambridge, England.

Palsberg, J. 1993. Correctness of binding-time analysis. *J. Funct. Program. 3,* 3 (July), 347–363.

Paulin-Mohring, C. and Werner, B. 1993. Synthesis of ML programs in the system Coq. *J. Symbol. Comput. 15*, 607–640.

Reynolds, J. C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference.* ACM Press, New York, 717–740.

Reynolds, J. C. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason, Ed. IFIP, Montvale, N.J., 513–523.

Ruf, E. 1993. Topics in online partial evaluation. Ph.D. thesis, Stanford Univ. , Stanford, Calif. Tech. Rep. CSL-TR-93-563.

Sestoft, P. 1989. Replacing function parameters by global variables. In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture*, J. E. Stoy, Ed. ACM Press, New York, 39–53.

Shivers, O. 1991. Control-flow analysis of higher-order languages or taming lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon Univ. Pittsburgh, Pa. Tech. Rep. CMU-CS-91-145.

Wadler, P. 1989. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture*, J. E. Stoy, Ed. ACM Press, New York, 347–359.

Wand, M. 1993. Specifying the correctness of binding-time analysis. *J. Funct. Program. 3,* 3 (July), 365–387.

WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. 1991. Automatic online partial evaluation. In *Proceedings of the 5th ACM Conference on Functional Programming and Computer Architecture*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 165–191.

Received August 1995; revised May 1996; accepted May 1996