The Essence of Eta-Expansion in Partial Evaluation

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

Abstract

Selective eta-expansion is a powerful "binding-time improvement", *i.e.*, a source-program modification that makes a partial evaluator yield better results. But like most binding-time improvements, the exact problem it solves and the reason why have not been formalized and are only understood by few.

In this paper, we describe the problem and the effect of eta-redexes in terms of monovariant binding-time propagation: eta-redexes preserve the static data flow of a source program by interfacing *static higher-order values in dynamic contexts* and *dynamic higher-order values in static contexts.* They contribute to two *distinct* binding-time improvements.

We present two extensions of Gomard's monovariant binding-time analysis for the pure λ-calculus. Our extensions annotate *and* eta-expand λ-terms. The first one eta-expands static higher-order values in dynamic contexts. The second also eta-expands dynamic higher-order values in static contexts.

As a significant application, we show that our first binding-time analysis suffices to reformulate the traditional formulation of a CPS transformation into a modern one-pass CPS transformer. This binding-time improvement is known, but it is still left unexplained in contemporary literature, *e.g.*, about "cps-based" partial evaluation.

We also outline the counterpart of eta-expansion for partially static data structures.

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

†College of Computer Science, 161 CN, 360 Huntington Avenue, Boston, MA 02115, USA. E-mail: palsberg@ccs.neu.edu

1 Introduction

Partial evaluation is a program-transformation technique for specializing programs [10, 15]. In the last decade it has been described using the notion of *binding times* [18]. Essentially the computations in a source program are divided into "static" or specialization-time computations (performed by the partial evaluator) and "dynamic" or run-time computations (to be performed in the specialized program). Partial evaluation amounts to performing the static computations and constructing the specialized program so that running it performs the dynamic computations. Thus a partial evaluator evaluates static expressions (*i.e.*, expressions that only depend on partial-evaluation time data) and reconstructs dynamic expressions (*i.e.*, expressions that depend on run-time data). For this to work, the binding-time division must be consistent [14, 21, 22], *i.e.*, no static computation may depend on the result of a dynamic computation.

In this setting, two sorts of expressible values coexist: static values and dynamic values (*i.e.*, residual expressions); correspondingly two sorts of contexts coexist as well: static contexts and dynamic contexts. (A context is an expression with one hole [1]. A higher-order (resp. partially static) context is an expression with a higher-order (resp. partially static) hole. A static (resp. dynamic) context is an expression with a static (resp. dynamic) hole.) To obtain consistency, Mix-style partial evaluators coerce static values and contexts to be respectively dynamic values and dynamic contexts, when they encounter a clash. This is acceptable if source programs are first-order and values are either fully static or fully dynamic. However these coercions are excessive for higher-order programs with partially static values and contexts.

Lacking better interface between higher-order and dynamic, one often must modify one's source programs "to improve their binding times" and thus "to make them specialize better". Jones, Gomard, and Sestoft [15, Section 12.4] list *eta-expansion* as an effective binding-time improvement but give only a brief idea of *why* it works.

To characterize the effect of eta-expansion, we will use the term *dynamize*, with the meaning "make dynamic". In the following section we explain how eta-redexes *prevent* a binding-time analysis from

- dynamizing static values in dynamic contexts, and
- dynamizing static contexts around dynamic values

when the values are higher-order. Preventing static values and contexts from being dynamized improves the annotation in case the static values are used elsewhere or in case other static values may also occur in the same context. In Section 3 we present two binding-time analyses that insert eta-redexes automatically, and we illustrate them with two continuation-based program transformations. Section 4 outlines the counterpart of eta-expansion for partially static data structures. After a comparison with related work, we conclude.

2 The essence of eta-expansion

We show three examples, where

- a number occurs both in a static and in a dynamic context,
- a *higher-order* value occurs both in a static and in a dynamic context, and
- a function is applied to both a static and a dynamic *higher-order* argument.

After the examples, we summarize why eta-expansion improves binding times, given a monovariant binding-time analysis.[1] We use "@" (pronounced "apply") to denote applications, and we abbreviate $(e_0@e_1)@e_2$ with $e_0@e_1@e_2$ and $e_0@(\lambda x.e)$ with $e_0@\lambda x.e$.

Reminder: eta-expanding a higher-order expression e yields the expression

$$\lambda v.e@v$$

where v does not occur free in e [1].

2.1 First-order static values in dynamic contexts

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

$$(\lambda x.(x+y)\times(x-1))@42$$

Assume that this β-redex will be reduced. The addition depends on the dynamic operand y, so it should be reconstructed (in other words, x occurs in a dynamic context). Both subtraction operands are static, so the subtraction can be performed (in other words, x occurs in a static context). The multiplication should be reconstructed since its first operand is dynamic. Overall, binding-time analysis yields the following two-level term.

$$(\overline{\lambda}x.(x\underline{+}y)\underline{\times}(x\overline{-}1))\overline{@}42$$

[1] A binding-time analysis is "monovariant" if it associates *one* binding-time description to any source expression. (It is "polyvariant" if it may associate several binding-time descriptions to any expression.) For consistency [14, 21, 22], in case of clash, a monovariant binding-time analysis approximates the clashing descriptions with an encompassing dynamic description, as illustrated in the following table.

binding-time description x	binding-time description y	least encompassing dynamic description of x and y
static	*dynamic*	*dynamic*
dynamic	*dynamic* $\rightarrow$ *dynamic*	*dynamic*
(*static*, *static*)	*dynamic*	*dynamic*
(*static*, *static*)	(*static*, *dynamic*)	(*static*, *dynamic*)
(*static*, *dynamic*)	(*dynamic*, *static*)	(*dynamic*, *dynamic*)

(Consistently with Nielson and Nielson [21], overlined means static and underlined means dynamic.)

We can summarize some of the binding-time information by giving the binding-time types of variables, as in Lambda-Mix [12, 15]. Here, x has type s (static) and y has type d (dynamic). After specialization (*i.e.*, two-level reduction), the residual term reads as follows.

$$(42+y)\times 41$$

The binding-time analysis is able to give an appropriate annotation of the above program because the argument to $\lambda x.(x+y)\times(x-1)$ is a *first-order* value. Inserting the static value in the dynamic context $([\cdot]+y)$ poses no problem. We now move on to the more difficult case where the inserted value is higher-order.

2.2 Higher-order static values in dynamic contexts

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

$$(\lambda f.f@g@f)@\lambda a.a$$

Again, assume that this β-redex is to be reduced. f occurs twice: once as the function part of an application (which here is a static context), and once as the argument of $f@g$ (which here is a dynamic context). The latter occurrence forces the binding-time analysis to classify f, and thus the rightmost λ-abstraction, to be dynamic (see Section 5 for a detailed motivation of this classification). Overall, binding-time analysis yields the following two-level term.

$$(\overline{\lambda}f.f\underline{@}g\underline{@}f)\overline{@}\underline{\lambda}a.a$$

Here, f has type d, and g has also type d. After specialization, the residual term reads as follows.

$$(\lambda a.a)@g@\lambda a.a$$

So unlike the first-order case, the fact that f, the static value, occurs in the dynamic context $f@g@[\cdot]$ "pollutes" its occurrence in the static context $[\cdot]@g@f$, so that neither is reduced.

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

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

$$(\lambda f.f@g@\boxed{\lambda y.f@y})@\lambda a.a$$

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

$$(\overline{\lambda}f.f\overline{@}g\underline{@}\underline{\lambda}y.f\overline{@}y)\overline{@}\overline{\lambda}a.a$$

Here, f has type $d\rightarrow d$, and both g and y have type d. Specialization yields the residual term

$$g@\lambda y.y$$

which is more reduced statically.

In this case, the eta-redex effectively protects the static higher-order expression $\lambda a.a$ from being dynamized in the remainder of the computation. Instead, only the occurrence in the dynamic context is affected.

2.3 Higher-order dynamic values in static contexts

The following expression is partially evaluated in a context where d_0 and d_1 are dynamic.

$$(\lambda f.f@d_0@(f@(\lambda x_1.x_1)@d_1))@\lambda a.a$$

f is applied twice: once to d_0 and something else, and once to $\lambda x_1.x_1$ and d_1. In a monovariant higher-order binding-time analysis, d_0 dynamizes $\lambda x_1.x_1$, since the first parameter of f can only have one binding time. Overall, binding-time analysis yields the following two-level term.

$$(\overline{\lambda} f.f\overline{@}d_0\underline{@}(f\overline{@}(\underline{\lambda} x_1.x_1)\underline{@}d_1))\overline{@\lambda} a.a$$

Here, f has type $d \rightarrow d$, x_1 has type d, and a has type d. Specialization yields the following residual term.

$$d_0@((\lambda x_1.x_1)@d_1)$$

In this situation, a binding-time improvement is possible since both d_0 and $\lambda x_1.x_1$ occur in a potentially-static context. We coerce d_0 by eta-expanding it (the eta-redex is boxed).

$$(\lambda f.f@\boxed{(\lambda x_0.d_0@x_0)}@(f@(\lambda x_1.x_1)@d_1))@\lambda a.a$$

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

$$(\overline{\lambda} f.f\overline{@}(\overline{\lambda} x_0.d_0\underline{@}x_0)\overline{@}(f\overline{@}(\overline{\lambda} x_1.x_1)\overline{@}d_1))\overline{@\lambda} a.a$$

Here, f has type $(d \rightarrow d) \rightarrow (d \rightarrow d)$, x_0 and x_1 both have type d, and a has type $d \rightarrow d$. Specialization yields the residual term

$$d_0@d_1$$

which is more reduced statically.

In this case, the eta-redex effectively prevents the dynamic expression d_0 from being propagated to f and dynamizing $\lambda x_1.x_1$ in the remainder of the computation. Instead, only the occurrence in the static context is affected.

2.4 Summary

In a monovariant binding-time analysis, each time a higher-order static value occurs both in a potentially static context and in a dynamic context, the dynamic context dynamizes the higher-order value, which in turn dynamizes the potentially static context.

Conversely, each time a higher-order static value and a dynamic value occur in a potentially static context, the dynamic value dynamizes the context, which in turn dynamizes the higher-order value.

Both problems can be circumvented by inserting eta-redexes in source programs. The eta-redex serves as "padding" around a value and inside a context, keeping one from dynamizing or from being dynamized by the other.

Eta-expanding a higher-order static expression f (when it occurs in a dynamic context) into

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

creates a value that can be used for replacement. This prevents the original expression from being dynamized by a dynamic context. Instead, the new abstraction is dynamized.

Eta-expanding a higher-order dynamic expression g (when it occurs in a potentially static context) into

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

creates a value that can be used for replacement. This prevents a potentially static context from being dynamized by g. Instead, the new application is dynamized.

Informally, eta-expansion changes the *two-level type* [21] of a term as follows. Assume that f and g have type $t_1 \rightarrow t_2$, where t_1 and t_2 are ground types. The first eta-expansion coerces the type $t_1 \overline{\rightarrow} t_2$ to be $t_1 \underline{\rightarrow} t_2$. The second eta-expansion coerces the type $t_1 \underline{\rightarrow} t_2$ to be $t_1 \overline{\rightarrow} t_2$. Note that *inside* the redexes, the type of f is still $t_1 \overline{\rightarrow} t_2$ and the type of g is still $t_1 \underline{\rightarrow} t_2$.

Further eta-expansion is necessary if t_1 or t_2 are not ground types. In fact, both kinds of eta-redex synergize. For example, if a higher-order static expression h has type $(t_1 \overline{\rightarrow} t_2) \overline{\rightarrow} t_3$ then its associated eta-redex reads as follows.

$$\underline{\lambda} v.h\overline{@\lambda} w.v\underline{@}w$$

In this example, the outer eta-expansion (of a static value in a dynamic context) creates the occurrence of a dynamic expression in a static context — hence the inner eta-redex.

To make our approach applicable to untyped languages, we will in the rest of the paper give dynamic entities the ground type d, as in Lambda-Mix [12, 15], rather than a two-level type such as $t_1 \underline{\rightarrow} t_2$.

Information to guide the insertion of eta-redexes can *not* be obtained directly from the output of a binding-time analysis: at that point all conflicts have been resolved. Moreover, it would be naïve to insert, say, one eta-redex around every subterm: sometimes more than one is needed for good results, as in the last example and in the CPS-transformation example in Section 3.3.2. Alternatively, programs could be required to be simply typed. Then the type of each subterm determines the maximal number of eta-redexes that might be necessary for that subterm. However this type-driven eta-redex insertion yields many unnecessary eta-redexes.

In the following section we demonstrate how to insert a small and appropriate number of eta-redexes automatically. Our approach does not require programs to be typed and both for the example in Section 2.2 and for Plotkin's CPS transformation we show that it gives a good result.

3 Automatic insertion of eta-redexes

3.1 Binding-time analysis for the pure λ-calculus

Our starting point is the binding-time analysis in Figure 1. The analysis is that of Gomard [12], restricted to the pure λ-calculus. Types are finite and generated from the following grammar.

$$t ::= d \mid t_1 \rightarrow t_2$$

The type d denotes the type of dynamic entities. The judgement $A \overset{old}{\vdash} e : t \triangleright w$ means that under hypothesis A, the λ-term e can be assigned type t with annotated term w.

3.2 Eta-expansion of static values in dynamic contexts

Figure 2 presents the first of our new binding-time analyses. It both inserts eta-redexes and binding-time annotates

$$A \overset{old}{\vdash} x : A(x) \rhd x$$

$$\frac{A[x \mapsto t_1] \overset{old}{\vdash} e : t_2 \rhd w}{A \overset{old}{\vdash} \lambda x.e : t_1 \to t_2 \rhd \overline{\lambda} x.w}$$

$$\frac{A[x \mapsto d] \overset{old}{\vdash} e : d \rhd w}{A \overset{old}{\vdash} \lambda x.e : d \rhd \underline{\lambda} x.w}$$

$$\frac{A \overset{old}{\vdash} e_0 : t_1 \to t_2 \rhd w_0 \qquad A \overset{old}{\vdash} e_1 : t_1 \rhd w_1}{A \overset{old}{\vdash} e_0 @ e_1 : t_2 \rhd w_0 \overline{@} w_1}$$

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

Figure 1: Gomard's binding-time analysis for the pure λ-calculus

$$A \vdash x : A(x) \rhd x \tag{1}$$

$$\frac{A[x \mapsto t_1] \vdash e : t_2 \rhd w}{A \vdash \lambda x.e : t_1 \to t_2 \rhd \overline{\lambda} x.w} \tag{2}$$

$$\frac{A[x \mapsto d] \vdash e : t_2 \rhd w \qquad t_2 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto t_2] \overset{old}{\vdash} m : d \rhd w'}{A \vdash \lambda x.e : d \rhd \underline{\lambda} x.w'[w/z]} \tag{3}$$

$$\frac{A \vdash e_0 : t_1 \to t_2 \rhd w_0 \qquad A \vdash e_1 : t_1 \rhd w_1}{A \vdash e_0 @ e_1 : t_2 \rhd w_0 \overline{@} w_1} \tag{4}$$

$$\frac{A \vdash e_0 : d \rhd w_0 \qquad A \vdash e_1 : t_1 \rhd w_1 \qquad t_1 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto t_1] \overset{old}{\vdash} m : d \rhd w_1'}{A \vdash e_0 @ e_1 : d \rhd w_0 \underline{@} (w_1'[w_1/z])} \tag{5}$$

z is always a fresh variable.

Figure 2: Binding-time analysis with eta-expansion of static values in dynamic contexts

$$d \vdash e \Rightarrow e \qquad\qquad \frac{t_1 \vdash x \Rightarrow x' \qquad t_2 \vdash e @ x' \Rightarrow e'}{t_1 \to t_2 \vdash e \Rightarrow \lambda x.e'}$$

Figure 3: Full eta-redex expansion

Consider the program $(\lambda f.f@g@f)@\lambda a.a$ from Section 2.2. We will now derive

$$\emptyset[g \mapsto d] \vdash (\lambda f.f@g@f)@\lambda a.a : \; d \triangleright (\overline{\lambda} f.f\overline{@}g\underline{@\lambda} y.f\overline{@}y)\overline{@\lambda} a.a.$$

Define $t = d \to d$ and $A = \emptyset[g \mapsto d][f \mapsto t]$. Consider the following fragment of the derivation, using rules 2 and 4.

$$\frac{\dfrac{A \vdash f@g@f : \; d \triangleright f\overline{@}g\underline{@\lambda} y.f\overline{@}y}{\emptyset[g \mapsto d] \vdash \lambda f.f@g@f : \; t \to d \triangleright \overline{\lambda} f.f\overline{@}g\underline{@\lambda} y.f\overline{@}y} \qquad \dfrac{\emptyset[g \mapsto d][x \mapsto d] \vdash x : \; d \triangleright x}{\emptyset[g \mapsto d] \vdash \lambda a.a : \; t \triangleright \overline{\lambda} a.a}}{\emptyset[g \mapsto d] \vdash (\lambda f.f@g@f)@\lambda a.a : \; d \triangleright (\overline{\lambda} f.f\overline{@}g\underline{@\lambda} y.f\overline{@}y)\overline{@\lambda} a.a}$$

We need to derive $A \vdash f@g@f : \; d \triangleright f\overline{@}g\underline{@\lambda} y.f\overline{@}y$. Here follows the last step of the derivation, using Rule 5.

$$\frac{A \vdash f@g : \; d \triangleright f\overline{@}g \qquad A \vdash f : \; t \triangleright f \qquad t \;\vdash\; z \;\Rightarrow\; \lambda y.z@y \qquad \emptyset[z \mapsto t] \overset{old}{\vdash} \lambda y.z@y : \; d \triangleright \underline{\lambda} y.z\overline{@}y}{A \vdash (f@g)@f : \; d \triangleright f\overline{@}g\underline{@\lambda} y.f\overline{@}y}$$

The last of the four assumptions is derived as follows.

$$\frac{\dfrac{\emptyset[z \mapsto t][y \mapsto d] \overset{old}{\vdash} z : \; t \triangleright z \qquad \emptyset[z \mapsto t][y \mapsto d] \overset{old}{\vdash} y : \; d \triangleright y}{\emptyset[z \mapsto t][y \mapsto d] \overset{old}{\vdash} z@y : \; d \triangleright z\overline{@}y}}{\emptyset[z \mapsto t] \overset{old}{\vdash} \lambda y.z@y : \; d \triangleright \underline{\lambda} y.z\overline{@}y}$$

Thus, our analysis inserts exactly the same eta-redex that we inserted by hand in Section 2.2.

Figure 4: Example of eta-expansion using the binding-time analysis of Figure 2

λ-terms. The judgement $A \vdash e : \; t \triangleright w$ means that under hypothesis A, the λ-term e can be assigned type t with annotated term w, where eta-redexes may have been inserted into w.

We use the judgement $t \;\vdash\; e \;\Rightarrow\; m$ to insert eta-redexes, see Figure 3. Intuitively, eta-redexes are inserted when the analysis meets a dynamic abstraction with a static body (Rule 3), and a dynamic application with a static argument (Rule 5). Only when the value is higher-order does eta-expansion takes place, see the first rule of Figure 3. When the value is first-order, eta-insertion is of course not possible. In the case of static values occurring in static contexts, it is not necessary.

Notice that our analysis generalizes Gomard's analysis: if in Rule 3 we always choose $t_2 = d$, and in Rule 5 we always choose $t_1 = d$, then we obtain Gomard's analysis.

If w is an annotated term, then $\hat{w}$ denotes the underlying λ-term. If $A \overset{old}{\vdash} \hat{w} : \; t \triangleright w$ for some A and t, then w is said to be well-annotated. A well-annotated term has a consistent binding-time division [22]. To prove that our analysis produces only well-annotated terms, we need the following lemma about Gomard's analysis.

Lemma 1 *Suppose z is the only free variable of e_2.*

$$\textit{If } A \overset{old}{\vdash} e_1 : \; t \triangleright w_1 \quad \textit{and} \quad \emptyset[z \mapsto t] \overset{old}{\vdash} e_2 : \; t' \triangleright w_2 \;,$$
$$\textit{then } A \overset{old}{\vdash} e_2[e_1/z] : \; t' \triangleright w_2[w_1/z] \;.$$

Proof. Consider the following more general property.

$$\text{If } A \overset{old}{\vdash} e_1 : \; t \triangleright w_1 \quad \text{and} \quad A' \overset{old}{\vdash} e_2 : \; t' \triangleright w_2 \;,$$
$$\text{then } A'' \overset{old}{\vdash} e_2[e_1/z] : \; t' \triangleright w_2[w_1/z] \;,$$

where A can be obtained from A'' by removing the binding for the free variables of e_2 except the one for z, and where A' can be obtained from A'' removing the bindings in A and adding the binding $z \mapsto t$. From this property the lemma immediately follows. The general property is proved by induction on the structure of the proof of $A' \overset{old}{\vdash} e_2 : t' \triangleright w_2$. □

We can then prove correctness: our analysis produces only well-annotated terms.

Theorem 2 *If $A \vdash e : \; t \triangleright w$, then $A \overset{old}{\vdash} \hat{w} : \; t \triangleright w$.*

Proof. By induction on the structure of the proof of $A \vdash e : t \triangleright w$, using Lemma 1 for Rules 3 and 5. □

As a corollary we get that even though eta-expansion allows more static reductions to take place, specialization will terminate since types are finite. In another setting, eta-expansion and generalization are used both to improve binding times and to ensure termination [6, 19, 20].

Future work includes finding an efficient implementation of the binding-time analysis.

3.3 Examples

3.3.1 Higher-order static values in dynamic contexts

In Figure 4 we demonstrate that the new binding-time analysis inserts the expected eta-redex in the example program from Section 2.2.

$$
\begin{array}{rcl}
[\![.]\!] & : & \text{syntax} \underline{\rightarrow} \text{syntax} \\
[\![x]\!] & = & \underline{\lambda} k.k \underline{@} x \\
[\![\lambda x.e]\!] & = & \underline{\lambda} k.k \underline{@ \lambda} x.[\![e]\!] \\
[\![e_0 @ e_1]\!] & = & \underline{\lambda} k.[\![e_0]\!] \underline{@ \lambda} v_0.[\![e_1]\!] \underline{@ \lambda} v_1.v_0 \underline{@} v_1 \underline{@} k
\end{array}
$$

$[\![e]\!]$ is the CPS counterpart of the expression e.

Figure 5: Two-level formulation of Plotkin's CPS transformation

$$
\begin{array}{rcl}
[\![.]\!] & : & (\text{syntax} \overline{\rightarrow} \text{syntax}) \overline{\rightarrow} \text{syntax} \\
[\![x]\!] & = & \overline{\lambda} k.k \overline{@} x \\
[\![\lambda x.e]\!] & = & \overline{\lambda} k.k \overline{@} \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v \\
[\![e_0 @ e_1]\!] & = & \overline{\lambda} k.[\![e_0]\!] \overline{@ \lambda} v_0.[\![e_1]\!] \overline{@ \lambda} v_1.v_0 \underline{@} v_1 \underline{@ \lambda} v_2.k \overline{@} v_2
\end{array}
$$

$\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v$ is the CPS counterpart of the expression e.

Figure 6: Two-level formulation of Plotkin's CPS transformation after eta-expansion

Define $A = \emptyset[[\![e]\!] \mapsto (d \rightarrow d) \rightarrow d][k \mapsto d \rightarrow d]$. Consider the following fragment of the derivation, using Rules 2 and 4.

$$
\dfrac{\dfrac{A \vdash k : \; d \rightarrow d \; \triangleright k \qquad A \vdash \lambda x.[\![e]\!] : \; d \; \triangleright \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v}{A \vdash k @ \lambda x.[\![e]\!] : \; d \; \triangleright k \overline{@} \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v}}{\emptyset[[\![e]\!] \mapsto (d \rightarrow d) \rightarrow d] \vdash \lambda k.k @ \lambda x.[\![e]\!] : \; (d \rightarrow d) \rightarrow d \; \triangleright \overline{\lambda} k.k \overline{@} \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v}
$$

We need to derive $A \vdash \lambda x.[\![e]\!] : \; d \; \triangleright \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v$. Define $t = (d \rightarrow d) \rightarrow d$ and $E = \lambda k.z @ (\lambda v.k @ v)$. Here follows the last step of the derivation, using Rule 3.

$$
\dfrac{A[x \mapsto d] \vdash [\![e]\!] : \; t \; \triangleright [\![e]\!] \qquad t \; \vdash \; z \; \Rightarrow \; E \qquad \emptyset[z \mapsto t] \overset{old}{\vdash} E : \; d \; \triangleright \underline{\lambda} k.z \overline{@ \lambda} v.k \underline{@} v}{A \vdash \lambda x.[\![e]\!] : \; d \; \triangleright \underline{\lambda} x.\underline{\lambda} k.[\![e]\!] \overline{@ \lambda} v.k \underline{@} v}
$$

Figure 7: Derivation in the case of abstraction

$$
\begin{array}{rcl}
[\![.]\!] & : & \text{syntax} \underline{\rightarrow} \text{syntax} \\
[\![x]\!] & = & \underline{\lambda} k.k \underline{@} x \\
[\![\lambda x.e]\!] & = & \underline{\lambda} k.k \underline{@ \lambda} x.[\![e]\!] \\
[\![e_0 @ e_1]\!] & = & \underline{\lambda} k.[\![e_0]\!]' \overline{@} \underline{\lambda} v_0.[\![e_1]\!]' \overline{@} \underline{\lambda} v_1.v_0 \underline{@} v_1 \underline{@} k \\
 & & \\
[\![.]\!]' & : & \text{syntax} \overline{\rightarrow} \text{syntax} \\
[\![x]\!]' & = & \overline{\lambda} k.k \underline{@} x \\
[\![\lambda x.e]\!]' & = & \overline{\lambda} k.k \underline{@ \lambda} x.[\![e]\!] \\
[\![e_0 @ e_1]\!]' & = & \overline{\lambda} k.[\![e_0]\!]' \overline{@} \underline{\lambda} v_0.[\![e_1]\!]' \overline{@} \underline{\lambda} v_1.v_0 \underline{@} v_1 \underline{@} k
\end{array}
$$

$[\![e]\!]$ is the CPS counterpart of the expression e.

Figure 8: Two-level formulation of Plotkin's CPS transformation with duplication

$$A \vdash x : A(x) \rhd x \quad (6)$$

$$\frac{A[x \mapsto t_1] \vdash e : t_2 \rhd w}{A \vdash \lambda x.e : t_1 \to t_2 \rhd \overline{\lambda} x.w} \quad (7)$$

$$\frac{A[x \mapsto t_1] \vdash e : d \rhd w \qquad t_2 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto d] \overset{old}{\vdash} m : t_2 \rhd w'}{A \vdash \lambda x.e : t_1 \to t_2 \rhd \overline{\lambda} x.w'[w/z]} \quad (8)$$

$$\frac{A[x \mapsto d] \vdash e : t_2 \rhd w \qquad t_2 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto t_2] \overset{old}{\vdash} m : d \rhd w'}{A \vdash \lambda x.e : d \rhd \underline{\lambda} x.w'[w/z]} \quad (9)$$

$$\frac{A \vdash e_0 : t_1 \to t_2 \rhd w_0 \qquad A \vdash e_1 : t_1 \rhd w_1}{A \vdash e_0 @ e_1 : t_2 \rhd w_0 \overline{@} w_1} \quad (10)$$

$$\frac{A \vdash e_0 : t_1 \to t_2 \rhd w_0 \qquad A \vdash e_1 : d \rhd w_1 \qquad t_1 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto d] \overset{old}{\vdash} m : t_1 \rhd w_1'}{A \vdash e_0 @ e_1 : t_2 \rhd w_0 \overline{@} (w_1'[w_1/z])} \quad (11)$$

$$\frac{A \vdash e_0 : d \rhd w_0 \qquad A \vdash e_1 : t_1 \rhd w_1 \qquad t_1 \vdash z \Rightarrow m \qquad \emptyset[z \mapsto t_1] \overset{old}{\vdash} m : d \rhd w_1'}{A \vdash e_0 @ e_1 : d \rhd w_0 \underline{@} (w_1'[w_1/z])} \quad (12)$$

z is always a fresh variable.

Figure 9: Binding-time analysis with both eta-expansion of static values in dynamic contexts and eta-expansion of dynamic values in static contexts

3.3.2 The CPS transformation

Let us now turn to the transformation of λ-terms into continuation-passing style (CPS). This example is significant because historically, the virtue of eta-redexes became apparent in connection with partial evaluation of CPS interpreters and with CPS transformers [2, 11]. Figure 5 displays Plotkin's original CPS transformation for the call-by-value lambda-calculus [23], written as a two-level term.

Since the transformation is a syntax constructor, *all* occurrences of @ and λ are dynamic. As a matter of fact, Gomard's binding-time analysis does classify all occurrences to be dynamic.

But CPS terms resulting from this transformation contain redundant "administrative" beta-redexes, which have to be post-reduced [26]. These beta-redexes can be avoided by inserting eta-redexes in the CPS transformation, allowing some beta-redexes in the transformation to become static.[2]

Figure 6 shows the revised transformation containing three extra eta-redexes: one for the CPS transformation of applications, and two for the CPS transformation of abstractions.

As analyzed elsewhere [11, Section 2], the eta-redex $\lambda k.[\![e]\!]@k$ prevents the outer $\lambda k....$ from being dynamized. The two other eta-redexes $\lambda v.k@v$ and $\lambda v_2.k@v_2$ enable k to be kept static. The types of the transformations (shown in the figures) summarize the binding-time improvement.

[2] In fact, in the particular case of the call-by-value CPS transformation, these static beta-redexes *precisely* coincide with Plotkin's administrative redexes [11]. However, this coincidence only happens for call-by-value and not, *e.g.*, for the call-by-name CPS transformation — an observation independently made in fall 1993 by John Hatcliff at Kansas State University and by Ray McDowell at the University of Pennsylvania (personal communication to the first author).

Our new analysis inserts exactly these three eta-redexes, given Plotkin's original specification. Figure 7 shows the derivation for the case of abstraction. The other two cases are left to the reader.

3.3.3 Improved "cps-based" cogen

Bondorf and Dussart's new work [5] relies on two key eta-expansions that are analogous to the ones of Section 3.3.2. These eta-expansions come for free with the binding-time analysis of Figure 2.

3.4 Eta-expansion of dynamic values in static contexts

Figure 9 presents the second of our new binding-time analyses. It is an extension of the binding-time analysis in Figure 1. Again, the judgement $A \vdash e : t \rhd w$ means that under hypothesis A, the λ-term e can be assigned type t with annotated term w, where eta-redexes may have been inserted into w.

Intuitively, eta-redexes are inserted when the analysis meets a static abstraction with a dynamic body (Rule 8), a dynamic abstraction with a static body (Rule 9), a static application with a dynamic argument (Rule 11), and a dynamic application with a static argument (Rule 12).

Again, the analysis generalizes Gomard's analysis: if we never use Rule 8 and Rule 11, and in Rule 9 we always choose $t_2 = d$, and in Rule 12 we always choose $t_1 = d$, then we obtain Gomard's analysis.

Theorem 2 holds also for this analysis, so the analysis produces only well-annotated terms.

Again, future work includes finding an efficient implementation of the binding-time analysis.

4 Partially static data structures

For data structures, we present a technique similar to eta-redexes that maintains the static data flow of source programs. As a prototypical example, we consider pairing. The ideas extend to other data structures in a straightforward manner.

Essentially, we delta-expand an expression p into the following expression.[3]

$$(\pi_1\, p, \pi_2\, p)$$

4.1 Partially static values in dynamic contexts

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

$$(\lambda p.g@(10 + \pi_1\, p)@p)@(1, d)$$

Again, let us assume that this β-redex is to be reduced. p occurs twice: once as the argument of the first projection π_1 (a static context), and once as the argument part of an application (a dynamic context). The latter occurrence forces the binding-time analysis to classify p, and thus the occurrence of π_1, to be dynamic. Overall, binding-time analysis yields the following two-level term.

$$(\overline{\lambda} p.g\underline{@}(10 \underline{+} \underline{\pi_1}\, p)\underline{@}p)\overline{@}\underline{(}1\underline{,}d\underline{)}$$

After specialization, the residual term reads as follows.

$$g@(10 + \pi_1\,(1, d))@(1, d)$$

The fact that p, the partially static value, occurs in a dynamic context "pollutes" its occurrence in the partially static context, so that neither are reduced.

NB: Since p occurs twice, a cautious binding-time analysis would reclassify the outer application to be dynamic, for fear of duplicating residual code. In that case, the expression is totally dynamic and thus is not simplified at all.

In this situation, a binding-time improvement is possible since $(1, d)$ will occur in a dynamic context. We can coerce this occurrence by inserting a delta-redex in the dynamic context (the redex is boxed).

$$(\lambda p.g@(10 + \pi_1\, p)@\boxed{(\pi_1\, p, \pi_2\, p)})@(1, d)$$

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

$$(\overline{\lambda} p.g\underline{@}(10 \overline{+} \overline{\pi_1}\, p)\overline{@}\underline{(}\overline{\pi_1}\, p\underline{,}\overline{\pi_2}\, p\underline{)})\overline{@}\overline{(}1\overline{,}d\overline{)}$$

Specialization yields the residual term

$$g@11@(1, d)$$

which is more reduced statically.

In this case, the delta-redex effectively prevents the partially static expression from being dynamized in the remainder of the computation. Instead, only the occurrence in the dynamic context is affected.

[3] Primitive operations are known as "delta rules" in the lambda-calculus [1].

4.2 Dynamic values in partially static contexts

It is simple to construct an example analogous to Section 2.3. Just have a partially static value and a dynamic value coexist in a context: the dynamic value dynamizes the context, which in turn, dynamizes the partially static value. Delta-expanding the corresponding dynamic expression in the source program prevents this approximation for the same (monovariant) binding-time analysis.

5 Related work

Our work is concerned with binding-time improvements and thus *off-line* partial evaluation of procedural programs. Eta-expansion is specific to the λ-calculus. We are not aware of any counterpart in partial evaluation of logic programs.

5.1 Mix-style partial evaluation

Mix-style partial evaluators developed at DIKU, such as Similix and Lambda-Mix [2, 12, 15], process procedural programs. When Similix-2 was developed [2], it was observed that the simple syntax reconstruction of Similix-1 [4] led to the occurrence of Scheme closures in residual programs. Two solutions were possible:

- lifting closures into syntax to construct the residual program (at specialization time); and
- dynamizing closures occurring in dynamic contexts (at binding-time analysis time).

The latter solution — coercing static values and contexts to be respectively dynamic values and dynamic contexts, in case of binding-time clash — was chosen in Similix-2, to avoid potential code duplication.[4] Thus one is forced

[4] For example, Similix specializes the source program

```
(define (main1 d)
  ((lambda (f)
     (cons f (f 2)))
   (lambda (a)
     (- (* 3 (- a d)) 1))))
```

into the following residual program.

```
(define (main1-0 d_0)
  (let ([f_2 (lambda (a_1)
               (- (* 3 (- a_1 d_0)) 1))])
    (cons f_2 (f_2 2))))
```

The decision to residualize is symptomatic of the tension inherent in partial evaluation: unfolding calls exposes opportunities for static computation, but if there are not many opportunities, or if they do not amount to much, unfolding just leads to code duplication. For example, Similix transforms the source program

```
(define (main2 d)
  ((lambda (f)
     (cons (f 1) (f 2)))
   (lambda (a)
     (- (* 3 (- a d)) 1))))
```

into the following residual program.

```
(define (main2-0 d_0)
  (cons (- (* 3 (- 1 d_0)) 1)
        (- (* 3 (- 2 d_0)) 1)))
```

to state this code duplication explicitly by garnishing one's source programs with eta-redexes (see [15, Section 10.1.4, Item (2)] and [15, Section 12.4] for two separate explanations). This solution has been consistently maintained in the later versions of Similix [3, 5], and adopted in Lambda-Mix [12, 15].

It seems that this decision, together with the forward nature of binding-time analysis [9], have created the need for binding-time improvements:

- Eta-expansion prevents the dynamization of higher-order values and contexts. Delta-expansion prevents the dynamization of partially-static values and contexts. Thus they both improve the binding times of source programs.
- Tail-recursive style in general (typically CPS) prevents the dynamization of intermediate results [9]. Thus it improves the static data flow of source programs.

5.2 Schism

Schism [8] does not dynamize static values whenever they occur in dynamic contexts. Instead, it inserts a "freeze" annotation coercing their result to be dynamic. For example, the term of Section 2.2 would be annotated as follows

$$(\overline{\lambda} f. f \overline{@} g \underline{@} (\text{freeze } f)) \overline{@ \lambda} a.a$$

and its specialization would yield the same good result as in Section 2.2. The freeze operator acts like eta-expansion, and enables Schism to deal with higher-order and partially static values in dynamic contexts without dynamizing them. Independently, Schism's polyvariant binding-time analysis [7] deals with dynamic values in static contexts, though currently, higher-order parameters are treated in a monovariant way.

Thus Schism's extra power makes it possible to interface partially static and higher-order values and contexts smoothly, without loss of static information. Eta-expansion enables such a smooth interface for Mix-style partial evaluators — *i.e.*, essentially, for the two-level λ-calculus [21], which is simpler. The two-level λ-calculus also proves particularly useful for specifying CPS transformations — something that was done so far by sheer insight [27] or by hand [11]. This mode of specification has direct applications to continuation-based program transformation [5, 16, 17].

5.3 Polyvariance and duplication

A polyvariant binding-time analysis maintains several binding-time descriptions for each source expression. The standard alternative is to duplicate source expressions. Figure 8 illustrates the effect of duplication as obtained by a polyvariant binding-time analysis whose target language is the two-level λ-calculus. Two variants are generated that are approximations of each other.[5] In this example, polyvariance does not yield the same effect as eta-expansion. This of course suggests to mix both — a future work.

[5] Polyvariance may be criticized for duplication but, as pointed out by one of the referees, this is only BTA-time duplication.

5.4 Online partial evaluation

An online partial evaluator such as FUSE [28] is inherently polyvariant over binding times [24] and thus meets no problem when dynamic values reach static contexts. The converse situation can be handled in specializers that carry two representations of each closure. Such systems include FUSE and Schism.

6 Conclusion

Inserting eta-redexes in source programs has until now been listed as black magic in the literature on Mix-style partial evaluation [15, Section 12.4]. We have described the effect of eta-redexes in terms of binding-time coercions: eta-redexes offer a syntactic representation of coercions and thus they prevent the binding-time analysis from approximating higher-order values and higher-order contexts to be dynamic. This effect is of prime importance for contemporary monovariant binding-time analyses since it enables one to carry out specialization as two-level β-reduction. It is also useful for contemporary polyvariant binding-time analyses, since eta-redexes can reduce the multiplication of variants. We also demonstrated how to integrate eta-expansion in an offline partial evaluator, by extending an existing binding-time analysis. Finally, we have outlined the counterpart of eta-expansion for partially static data structures.

Future work naturally includes developing a partial evaluator with better coercions, to eliminate the need of binding-time improvements by eta-expansion.

Acknowledgements

We are grateful to Andrzej Filinski, Neil Jones, Julia Lawall, Torben Mogensen, and the referees for insightful comments. The first author also thanks Charles Consel for fundamental discussions about the nature of partial evaluation.

References

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

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

[3] 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.

[4] 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.

[5] Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-*

Based Program Manipulation, Orlando, Florida, June 1994.

[6] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 308–317, Copenhagen, Denmark, June 1993. ACM Press.

[7] Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [25], pages 66–77.

[8] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Schmidt [25], pages 145–154.

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

[10] 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.

[11] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[12] Carsten K. Gomard. *Program Analysis Matters*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, November 1990. DIKU Report 91-17.

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

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

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

[16] Julia L. Lawall. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[17] 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*, Orlando, Florida, June 1994. ACM Press.

[18] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, March 1989.

[19] Torben Æ. Mogensen. Constructor specialization. In Schmidt [25], pages 22–32.

[20] Christian Mossin. Partial evaluation of general parsers. In Schmidt [25], pages 13–21.

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

[22] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(32):347–363, 1993.

[23] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

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

[26] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[27] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311, Pittsburgh, Pennsylvania, March 1991. 7th International Conference.

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