Eta-Redexes in Partial Evaluation

Jens Palsberg

Purdue University, Dept of Computer Science
W Lafayette, IN 47907, palsberg@cs.purdue.edu

Abstract. Source-program modifications can make a partial evaluator yield dramatically better results. For example, eta-redexes can preserve static data flow by acting as an interface between values and contexts. This note presents a type-based explanation of what eta-expansion achieves, why it works, and how it can be automated. This leads to a unified view of various source-code improvements, including a popular transformation called "The Trick."

1 Introduction

This note presents ideas from papers by Danvy, Malmkjær, and Palsberg [9, 10].

1.1 Background

Partial evaluation is a program-transformation technique for specializing programs [6, 14]. 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). 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 computation depends on the result of a dynamic computation, and (2) no static parts are left in the residual program [13, 19, 20, 25]. 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. In an offline partial evaluator, the precision of the binding-time analysis determines the effectiveness of the program specializer [6, 14]. 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.

Recall that a context is an expression with one hole [1]. A static (resp. dynamic) context is an expression with one hole where the expression fitting this hole is static (resp. dynamic).

To obtain consistency, Mix-style partial evaluators [14] 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 programs with partially static values and contexts.

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 [14, Ch. 12] documents three such "binding-time improvements": continuation-passing style, eta-expansion, and "The Trick." In this note we present the basic idea which in [9, 10] is fully developed into a unified view of these binding-time improvements.

1.2 Eta-expansion

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 [1]. By analogy, "eta-expanding" a product expression e of type $\tau_1 \times \tau_2$ yields the expression

$$\mathit{Pair}(\mathit{Fst}\, e, \mathit{Snd}\, e)\ ,$$

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

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

In the following section we will show how eta-expansion *prevents* a binding-time analysis from

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

when the values are of function, product, or disjoint-sum type. (We use "dynamize" to mean "make dynamic.) Preventing static values and contexts from being dynamized improves the annotation in cases where the static values are used elsewhere or in cases where other static values may also occur in the same context. Thus, eta-expansion serves as a binding-time coercion for static values in dynamic contexts and for dynamic values in potentially static contexts. We can view an eta-redex as providing a syntactic representation of a binding-time coercion, either from static to dynamic, or from dynamic to static. Eta-expansion can also help ensure termination of a partial evaluator [5, 17, 18].

In the case of using eta-expansion where the value is of disjoint-sum type, the binding-time improvement enables "The Trick." Intuitively, "The Trick" 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 reduce 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 [14, Sec. 12.2]. The Trick is usually carried out explicitly by the programmer (see the while loop in Jones and Gomard's Imperative Mix [14, Sec. 4.8.3]).

This enumeration of static values could also be obtained by program analysis, for example using Heintze's set-based analysis [12]. 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 [24] (also known as closure analysis [23]) 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 [22] and recently cast in a typed setting [16]. Since the end of the eighties, it is used by such partial evaluators as Similix to handle higher-order programs [3]. The conclusion of this is that Jones, Gomard, and Sestoft actually do use an automated version of The Trick [14, Sec. 10.1.4, Item (1)], even if they do not present it as such.

In this note we concentrate on eta-expansion of expressions of function type. In the following section we give several examples of what eta-expansion achieves and why it works. In Section 3 we present a binding-time analysis that inserts eta-redexes automatically using two coercion rules, and we show that it transforms Plotkin's CPS-transformation into the improved form studied by Danvy and Filinski [8].

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. We use "@" (pronounced "apply") to denote applications, and we abbreviate $(e_0@e_1)@e_2$ by $e_0@e_1@e_2$.

2.1 First-order static values in dynamic contexts

The following expression is partially evaluated in a context with y 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, $[\cdot] + y$). Both subtraction operands are static, so the subtraction can be performed (in other words, x occurs in a static context, $[\cdot]-1$). The multiplication should be reconstructed since its first operand is dynamic. Overall, binding-time analysis yields the following two-level term.

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

(Consistently with Nielson and Nielson [19], 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 [11, 14]. 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.

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

Lambda-Mix's 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 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 with g 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. 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 statically.

NB: Since f 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 $\lambda a.a$ 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 $d \to d$, 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 with d_0 and d_1 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 once to $\lambda x_1.x_1$. 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.

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

Here, f has type $d \to d$, x_1 has type d, and a has type d (corresponding to the type of d_0). Specialization yields the following residual term.

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

The context $f@[\cdot]$ occurs twice in the source term. The dynamic value d_0 appears in the first occurrence, and the static value $\lambda x_1.x_1$ appears in the second occurrence. Since the context can only have one binding time (since it is the same f), d_0 pollutes $f@[\cdot]$, which in turn pollutes $\lambda x_1.x_1$. Since f is in fact $\lambda a.a$, the result of the application becomes dynamic. So the two potentially static applications of this result, respectively $[\cdot]@(f@(\lambda x_1.x_1)@d_1)$ and $[\cdot]@d_1$, become dynamic.

In this situation, a binding-time improvement is possible since both d_0 and $\lambda x_1.x_1$ occur (as results) 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.

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

Here, f has type $(d \to d) \to (d \to d)$, corresponding to statically applying f to both arguments. x_0 and x_1 both have type d, and a has type $d \to d$ (corresponding to the type of $\lambda x_1.x_1$). 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 the same 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* [19] of a term as follows. Assume that f and g have type $t_1 \to t_2$, 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

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

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 [11, 14], 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 3.2. 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. Such type-driven eta-redex insertion is closely related to Danvy's type-directed partial evaluation [7].

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 The binding-time analysis

Our binding-time analysis is specified in Figure 1. It can easily be extended to handle products and sums. Details of this together with a correctness proof are given in [10].

The definition in Figure 1 has three parts. The first part is the binding-time analysis of Gomard [11], 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 judgment $A \vdash e : t \triangleright w$ means that under hypothesis A, the λ-term e can be assigned type t with annotated term w.

The second part of our analysis is two rules for binding-time coercion. Intuitively, the two rules can be understood as being able (1) to coerce the binding-time type d to any type τ and (2) to coerce any type τ to the type d. 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 the third part of our analysis. 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 [2, 7].

Consider the first binding-time coercion rule in Figure 1. Intuitively, it works as follows. We are given a λ-term e that we would like to assign the type τ. In

Gomard's binding-time analysis:

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

$$\frac{A[x \mapsto t_1] \vdash e : t_2 \triangleright w}{A \vdash \lambda x.e : t_1 \to t_2 \triangleright \overline{\lambda} x.w} \qquad \frac{A[x \mapsto d] \vdash e : d \triangleright w}{A \vdash \lambda x.e : d \triangleright \underline{\lambda} x.w}$$

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

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

Rules for binding-time coercion:

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

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

Rules for eta-expansion:

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

Fig. 1. The binding-time analysis.

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 binding-time coercion rule in Figure 1 works in a similar way.

Our binding-time analysis inserts the expected eta-redex in the example program $(\lambda f.f@g@f)@(\lambda a.a)$ from Section 2.2:

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

where t abbreviates $d \to d$ and A abbreviates $\emptyset[g \mapsto d][f \mapsto t]$.

Here follows part of the derivation of $A \vdash f : d \triangleright \underline{\lambda} y.f\overline{@}y$.

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

Notice that $\underline{\lambda} y.f\overline{@}y$ is dynamic while f remain static.

3.2 Example: 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 [3, 8]. It also has practical interest since the pattern of construction and use of higher-order values in the CPS transform is prototypical. The first part of Figure 2 displays Plotkin's original CPS transformation for the call-by-value lambda-calculus [21], written as a two-level term.

Before eta-expansion, where $[\![e]\!]$ is the CPS-version of the expression e:

$$\begin{aligned}
[\![.]\!] &: \text{syntax} \underline{\Rightarrow} \text{CPSsyntax} \underline{\rightarrow} \text{CPSsyntax} \\
[\![x]\!] &= \underline{\lambda} k.k\underline{@}x \\
[\![\lambda x.e]\!] &= \underline{\lambda} k.k\underline{@}(\underline{\lambda} x.[\![e]\!]) \\
[\![e_0@e_1]\!] &= \underline{\lambda} k.[\![e_0]\!]\underline{@}(\underline{\lambda} v_0.[\![e_1]\!]\underline{@}(\underline{\lambda} v_1.(v_0\underline{@}v_1)\underline{@}k))
\end{aligned}$$

After eta-expansion, where $\underline{\lambda} k.[\![e]\!]\overline{@}\overline{\lambda} v.k\underline{@}v$ is the CPS-version of the expression e:

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

Fig. 2. Two-level formulation of Plotkin's CPS transformation.

Since the transformation is a syntax constructor, *all* occurrences of @ and λ are dynamic. And in 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 [15]. These beta-redexes can be avoided by inserting eta-redexes in the CPS transformation, allowing some beta-redexes in the transformation to become static.

The second part of Figure 2 shows the revised transformation containing three extra eta-redexes: one for the CPS transformation of applications, and two for the CPS transformation of abstractions (the eta-redexes are boxed.)

As analyzed by Danvy and Filinski [8], 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.

Our new analysis inserts exactly these three eta-redexes, given Plotkin's original specification. Here follows part of the derivation for the case of abstraction. We use the abbreviations $t = (d \to d) \to d$ and $A = \emptyset[[\![e]\!] \mapsto t][k \mapsto d \to d]$.

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

We need to derive $A \vdash \lambda x.[\![e]\!] : d \triangleright \underline{\lambda} x.\underline{\lambda} k.[\![e]\!]\overline{@}(\overline{\lambda} v.k\underline{@}v)$. We use the abbreviation $E = \lambda k.z@(\lambda v.k@v)$. Here follows the last two steps of the derivation.

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

Bondorf and Dussart's work [4] relies on two key eta-expansions that are analogous to those studied in this section.

References

1. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
2. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *LICS'91, Sixth Annual Symposium on Logic in Computer Science*, pages 203–211, 1991.
3. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34, December 1991.
4. Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–10, 1994.
5. Anders Bondorf and Jens Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996. Preliminary version in Proc. FPCA'93, Sixth ACM Conference on Functional Programming Languages and Computer Architecture, pages 308–317, Copenhagen, Denmark, June 1993.
6. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. POPL'93, Twentieth Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 493–501, 1993.
7. Olivier Danvy. Type-directed partial evaluation. In *Proc. POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 242–257, 1996.

8. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
9. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995. Preliminary version in Proc. PEPM'94, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 11–20, Orlando, Florida, June 1994.
10. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does the trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, November 1996.
11. Carsten K. Gomard. *Program Analysis Matters*. PhD thesis, DIKU, University of Copenhagen, November 1991. DIKU Report 91–17.
12. Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, October 1992. CMU–CS–92–201.
13. Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In *Proc. Partial Evaluation and Mixed Computation*, pages 225–282, 1988.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
15. 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.
16. Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 271–283, 1996.
17. Torben Æ Mogensen. Constructor specialization. In *Proc. PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, 1993.
18. Christian Mossin. Partial evaluation of general parsers. In *Proc. PEPM'93, Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–21, 1993.
19. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.
20. Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.
21. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
22. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. 25th ACM National Conference*, pages 717–740. ACM Press, 1972.
23. Peter Sestoft. Replacing function parameters by global variables. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, 1989.
24. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.
25. Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, 1993.