

A New Approach to One-Pass Transformations

Kevin Millikin

BRICS *

Department of Computer Science
University of Aarhus **

Abstract

We show how to construct a one-pass optimizing transformation by fusing a non-optimizing transformation with an optimization pass. We state the transformation in build form and the optimization pass in cata form, i.e., as a catamorphism; and we use cata/build fusion to combine them. We illustrate the method by fusing Plotkin’s call-by-value and call-by-name CPS transformations with a reduction-free normalization function for the λ -calculus, thus obtaining two new one-pass CPS transformations. The method can be used to construct other one-pass transformations via program fusion.

1 INTRODUCTION

Compiler writers often face a choice between implementing a simple, non-optimizing transformation pass that generates poor code which will require subsequent optimization, and implementing a complex, optimizing transformation pass that avoids generating poor code in the first place. A two-pass strategy is compelling because it is simpler to implement correctly, but its disadvantage is that the intermediate data structures can be large and traversing them unnecessarily can be costly. In a system performing just-in-time compilation or run-time code generation, the costs associated with a two-pass compilation strategy can render it impractical. A one-pass optimizing transformation is compelling because it avoids generating intermediate data structures requiring further optimization, but its disadvantage is that the transformation is more difficult to implement.

The specification of a one-pass transformation is that it is extensionally equal to the composition of a non-optimizing transformation and an optimization pass. A one-pass transformation is not usually constructed this way however, but is instead constructed as a separate artifact which must then be demonstrated to match its specification. Our approach is to directly construct one-pass transformations as the fusion of passes via shortcut deforestation [15, 25], thus maintaining the explicit connection to both the non-optimizing transformation and the optimization pass.

Shortcut deforestation relies on one of a pair of simple but powerful program transformations known as cata/build fusion and destroy/ana fusion.¹ Using destroy/ana fusion

*Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

**IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. Email: kmillikin@brics.dk

¹Svenningsson uses the term *destroy/unfoldr* [24] to refer the transformation that eliminates intermediate lists produced by the function *unfoldr*. We coin the term *destroy/ana* to refer to the same rule used to fuse arbitrary intermediate data structures.

to construct a one-pass transformation requires that the non-optimizing transformation be an *anamorphism*: a function whose call tree is isomorphic to the structure of the output it produces. Since we use the continuation-passing style (CPS) transformation as our example in this paper, and since the CPS transformation is not an anamorphism, we concentrate on cata/build fusion instead. The method described here would work equally well for fusing an anamorphic transformation with an optimization pass in destroy form.

Cata/build fusion requires both the transformation and optimization passes to be expressed in a stylized form. The transformation must be in *build form*, abstracted over the constructors of its input; and the optimization pass must be a *catamorphism*, defined by compositional recursive descent over its input.

The non-optimizing CPS transformation generates terms that contain *administrative redexes* which can be optimized away by β -reduction. A one-pass CPS transformation [10, 11] generates terms that do not contain administrative redexes, in a single pass, by contracting these redexes at transformation time. Thus β -reduction is the notion of optimization for the CPS transformation. The normalization function we will use for reduction of CPS terms, however, contracts all β -redexes, not just administrative redexes. In Section 6 we describe how to contract only the administrative redexes.

When using a metalanguage to express normalization in the object language, as we do here, the evaluation order of the metalanguage is usually important. However, because CPS terms are insensitive to evaluation order [22], evaluation order is not a concern.

This work. We present a systematic method to construct one-pass transformations, based on the fusion of a non-optimizing transformation with an optimization pass. We demonstrate the method by constructing new one-pass CPS transformations as the fusion of non-optimizing CPS transformations with a catamorphic normalization function.

The rest of the paper is organized as follows. First, we briefly review catamorphisms and cata/build fusion in Section 2. Then, in Section 3, we restate Plotkin’s call-by-value CPS transformation [22] in build form; and in Section 4 we restate a reduction-free normalization function for the untyped λ -calculus to use a catamorphism. We then present a new one-pass CPS transformation obtained by fusion, in Section 5. In Section 6 we describe how to modify the transformation to contract only the administrative redexes. We compare our new CPS transformation to the one-pass transformation of Danvy and Filinski [11] in Section 7. In Section 8 we repeat the method for Plotkin’s call-by-name CPS transformation; and we present related work and conclude in Section 9.

Prerequisites. The reader should be familiar with reduction in the λ -calculus, and the CPS transformation. Knowledge of functional programming, particularly catamorphisms (i.e., the higher-order function *fold*) is expected.

2 CATAMORPHISMS OVER λ -TERMS

An algebraic datatype is the least fixed point of a (category-theoretic) functor capturing the shape of the datatype. In particular, the familiar datatype of λ -terms (assuming a set

Ident of identifiers), defined by the following context-free grammar:

$$Term \ni m ::= \text{var } x \mid \text{lam } (x, m) \mid \text{app } (m, m)$$

is the least solution to the equation $X \cong TX$, where T is the functor:

$$TA = Ident + (Ident \times A) + (A \times A)$$

We use var_T , lam_T , and app_T for the injection functions into the sum on the right-hand side. T describes λ -terms abstracted over the type of their subterms. The action of T on a function $f : A \rightarrow B$ is to produce a new function $Tf : TA \rightarrow TB$ that applies f to the subparts of its input. The isomorphism $Term \cong T(Term)$ is witnessed by a pair of functions $in : T(Term) \rightarrow Term$ and $out : Term \rightarrow T(Term)$.

A pair $\langle A, \varphi \rangle$ of a type A and a function $\varphi : TA \rightarrow A$ (such as $\langle Term, in \rangle$) is called a T -algebra. A T -algebra homomorphism is a function $f : A \rightarrow B$ between the carriers of two T -algebras that makes the following diagram commute:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \uparrow & & \uparrow \\ TA & \xrightarrow{Tf} & TB \end{array}$$

T -algebras and T -algebra homomorphisms form a category, and $\langle Term, in \rangle$ is the initial object in this category. Initiality guarantees that for any T -algebra φ , there is a unique T -algebra homomorphism f that makes the following diagram commute:

$$\begin{array}{ccc} Term & \xrightarrow{f} & A \\ in \uparrow & & \uparrow \varphi \\ T(Term) & \xrightarrow{Tf} & TA \end{array}$$

The data T and φ uniquely determine f , which is known as a T -catamorphism (of φ) and written with “banana” brackets [21] as $\langle \varphi \rangle_T$.

Because in and out are inverses, the following diagram also commutes:

$$\begin{array}{ccc} Term & \xrightarrow{\langle \varphi \rangle_T} & A \\ out \downarrow & & \uparrow \varphi \\ T(Term) & \xrightarrow{T \langle \varphi \rangle_T} & TA \end{array}$$

The catamorphism $\langle \varphi \rangle_T$ can be defined as the least fixed point of a functional that uses out to decompose a λ -term into a $T(Term)$ which contains the arguments of the recursive calls as subparts, maps itself over the subparts to produce the results of the recursive calls, and uses the T -algebra φ to construct final the result:

$$\llbracket \varphi \rrbracket_T = \text{fix}(\lambda f. \varphi \circ T f \circ \text{out})$$

Cata/build fusion is a simple program transformation that fuses a catamorphism with a function written in build form. A function $f : A \rightarrow \text{Term}$ is in build form if it is written as a polymorphic function $\tau : \forall B. (TB \rightarrow B) \rightarrow A \rightarrow B$ (i.e., a function abstracted over an arbitrary T -algebra) applied to the initial T -algebra in . In that case, the acid-rain theorem allows fusion with a catamorphism $\llbracket \varphi \rrbracket_T$:

Theorem 1 (Acid rain for catamorphisms [25]). *If τ is a polymorphic function of type $\forall B. (TB \rightarrow B) \rightarrow A \rightarrow B$ then*

$$\llbracket \varphi \rrbracket_T \circ (\tau \text{ in}) = \tau \varphi$$

3 THE CALL-BY-VALUE CPS TRANSFORMATION IN BUILD FORM

The non-optimizing call-by-value CPS transformation [22] is given in Figure 1. We assume the ability to choose fresh identifiers when needed; the identifiers k , v_0 , and v_1 are chosen fresh.

$$\begin{aligned} \text{transform} & && : \text{Term} \rightarrow \text{Term} \\ \text{transform} (\text{var } x) & && = \text{lam } (k, \text{app } (\text{var } k, \text{var } x)) \\ \text{transform} (\text{lam } (x, m)) & && = \text{lam } (k, \text{app } (\text{var } k, \text{lam } (x, \text{transform } m))) \\ \text{transform} (\text{app } (m_0, m_1)) & && = \text{lam } (k, \text{app } (\text{transform } m_0, \\ & && \text{lam } (v_0, \text{app } (\text{transform } m_1, \\ & && \text{lam } (v_1, \text{app } (\text{app } (\text{var } v_0, \text{var } v_1), \text{var } k)))))) \end{aligned}$$

FIGURE 1. Plotkin's non-optimizing call-by-value CPS transformation

Fusion with a catamorphic normalization function requires that the transformation is put into build form, i.e., parameterized over the constructors used to produce its output. We first replace each of the λ -term constructors with the composition of the corresponding T -constructor followed by the initial T -algebra in (e.g., replace app with $\text{in} \circ \text{app}_T$), and then abstract over the T -algebra. The transformation in build form thus constructs a Church encoding of the original output,² with the functor T carrying the arguments to the λ -term constructors and the injection tags of T selecting between constructors. The non-optimizing transformation in build form is shown in Figure 2.

4 A CATAMORPHIC NORMALIZATION FUNCTION

Normalization by evaluation (NBE) is a reduction-free approach to normalization that is not based on the transitive closure of a single-step reduction function. Instead, NBE

²Swapping the order of the arguments to τ in Figure 2 yields a function that constructs the Church encoding of the output of transform in Figure 1

$$\begin{aligned}
\tau & : \forall A.(TA \rightarrow A) \rightarrow Term \rightarrow A \\
\tau \varphi (\text{var } x) & = \varphi (\text{lam}_T (k, \varphi (\text{app}_T (\varphi (\text{var}_T k), \varphi (\text{var}_T x)))))) \\
\tau \varphi (\text{lam } (x, m)) & = \varphi (\text{lam}_T (k, \varphi (\text{app}_T (\varphi (\text{var}_T k), \varphi (\text{lam}_T (x, \tau \varphi m)))))) \\
\tau \varphi (\text{app } (m_0, m_1)) & = \varphi (\text{lam}_T (k, \varphi (\text{app}_T (\tau \varphi m_0, \\
& \quad \varphi (\text{lam}_T (v_0, \varphi (\text{app}_T (\tau \varphi m_1, \\
& \quad \quad \varphi (\text{lam}_T (v_1, \varphi (\text{app}_T (\varphi (\text{app}_T (\varphi (\text{var}_T v_0), \varphi (\text{var}_T v_1))), \\
& \quad \quad \quad \varphi (\text{var}_T k)))))))))))))
\end{aligned}$$

$$\begin{aligned}
\text{transform} & : Term \rightarrow Term \\
\text{transform } m & = \tau \text{ in } m
\end{aligned}$$

FIGURE 2. Non-optimizing CPS transformation in build form

uses a non-standard evaluator to map a term to its denotation in a *residualizing model*. The residualizing model has the property that a denotation in the model can be reified into a syntactic representation of a term, and that reified terms are in normal form. A *reduction-free* normalization function is then constructed as the composition of evaluation and reification.

NBE has been used in the typed λ -calculus, combinatory logic, the free monoid, and the untyped λ -calculus [9]. We adopt the traditional normalization function for the untyped λ -calculus as our optimizer for CPS terms. We show it in Figure 3. Just as in the CPS transformation, we assume the ability to choose a fresh identifier when needed. We note, however, that our approach works equally well with other methods of name generation (e.g., threading a source of fresh names through the evaluator or using de Bruijn levels).

The normalization function maps terms to their β -normal form. Normal forms are given by the grammar for *Norm* in Figure 3. Elements *Val* of the residualizing model are either atoms (terms that are not abstractions, given by the grammar for *Atom*), or else functions from *Val* to *Val*.

Environments are somewhat unusual in that the initial environment maps each identifier to itself as an element of the residualizing model, which allows us to handle open terms:

$$\begin{aligned}
\rho_{init} x & = \text{res } (\text{var}_N x) \\
(\rho \{x \mapsto v\}) x & = v \\
(\rho \{y \mapsto v\}) x & = \rho x, \text{ if } x \neq y
\end{aligned}$$

Abstractions denote functions from *Val* to *Val*. The recursive function *reify* (\downarrow) extracts a normal term from an element of the residualizing model. The function *app* dispatches on the value of the operator of an application to determine whether to build a residual atom or to apply the function.

Normalization is then the composition of evaluation (in the initial environment) followed by reification. Because the evaluation function is compositional, we can rewrite it

$Norm \ni n$	$::= \text{atom}_N a \mid \text{lam}_N (x, n)$
$Atom \ni a$	$::= \text{var}_N x \mid \text{app}_N (a, n)$
Val	$= Atom + (Val \rightarrow Val)$
$Val \ni v$	$::= \text{res } a \mid \text{fun } f$
Env	$= Ident \rightarrow Val$
$Env \ni \rho$	$::= \rho_{init} \mid \rho\{x \mapsto v\}$
$eval$	$: Term \rightarrow Env \rightarrow Val$
$eval (\text{var } x) \rho$	$= \rho x$
$eval (\text{lam } (x, m)) \rho$	$= \text{fun } (\lambda v. eval m (\rho\{x \mapsto v\}))$
$eval (\text{app } (m_0, m_1)) \rho$	$= app (eval m_0 \rho) (eval m_1 \rho)$
\downarrow	$: Val \rightarrow Norm$
$\downarrow (\text{res } a)$	$= \text{atom}_N a$
$\downarrow (\text{fun } f)$	$= \text{lam}_N (x, \downarrow (f (\text{res } (\text{var}_N x))))$, where x is fresh
app	$: Val \rightarrow Val \rightarrow Val$
$app (\text{res } a) v$	$= \text{res } (\text{app}_N (a, \downarrow v))$
$app (\text{fun } f) v$	$= f v$
$normalize$	$: Term \rightarrow Norm$
$normalize m$	$= \downarrow (eval m \rho_{init})$

FIGURE 3. Reduction-free normalization function

as a catamorphism over λ -terms, given in Figure 4. The domains of terms, atoms, values, and environments do not change, nor do the auxiliary functions \downarrow and app .

5 A NEW ONE-PASS CALL-BY-VALUE CPS TRANSFORMATION

We fuse the non-optimizing CPS transformation $(\tau in) : Term \rightarrow Term$ of Section 3 and the catamorphic evaluation function $(\Downarrow)_T : Term \rightarrow Env \rightarrow Val$ of Section 4 to produce a one-pass transformation from λ -terms into the residualizing model. This one-pass transformation is simply $(\tau \Downarrow) : Term \rightarrow Env \rightarrow Val$. We then extract β -normal forms from the residualizing model by applying to the initial environment and reifying, as before.

Inlining the definitions of τ and \Downarrow , performing β -reduction, and simplifying environment operations (namely, replacing environment applications where the value is known with that value and trimming bindings that are known to be unneeded) yields the simplified specification of the one-pass transformation shown in Figure 5. The domains of normal terms, atoms, values, and environments as well as the auxiliary functions \downarrow and app are the same as in Figure 3.

We have implemented this one-pass transformation in Standard ML, letting the ML

$$\begin{aligned}
\varphi & & : T(Env \rightarrow Val) \rightarrow Env \rightarrow Val \\
\varphi(\text{var}_{\top} x) \rho & = \rho x \\
\varphi(\text{lam}_{\top}(x, m)) \rho & = \text{fun}(\lambda v. m(\rho\{x \mapsto v\})) \\
\varphi(\text{app}_{\top}(m_0, m_1)) \rho & = \text{app}(m_0 \rho)(m_1 \rho) \\
\\
\text{normalize} & & : Term \rightarrow Term \\
\text{normalize } m & = \downarrow((\varphi)_T m \rho_{init})
\end{aligned}$$

FIGURE 4. Normalization function with catamorphic evaluator

$$\begin{aligned}
xform & & : Term \rightarrow Env \rightarrow Val \\
xform(\text{var } x) \rho & = \text{fun}(\lambda k. \text{app } k(\rho x)) \\
xform(\text{lam}(x, m)) \rho & = \text{fun}(\lambda k. \text{app } k(\text{fun}(\lambda v. xform m(\rho\{x \mapsto v\})))) \\
xform(\text{app}(m_0, m_1)) \rho & = \text{fun}(\lambda k. \text{app}(xform m_0 \rho) \\
& \quad (\text{fun}(\lambda v_0. \text{app}(xform m_1 \rho) \\
& \quad \quad (\text{fun}(\lambda v_1. \text{app}(\text{app } v_0 v_1) k)))))) \\
\\
transform & & : Term \rightarrow Norm \\
transform m & = \downarrow(xform m \rho_{init})
\end{aligned}$$

FIGURE 5. A new one-pass call-by-value CPS transformation

inferencer acts as a theorem prover to verify that the transformation returns a β -normal form if it terminates [12].

6 SUPPRESSING CONTRACTION OF SOURCE REDEXES

Compared to traditional one-pass CPS transformations, our transformation is overzealous. The normalization function we use contracts all β -redexes; it cannot tell which ones are administrative redexes. Therefore our CPS transformation does not terminate for terms that do not have a β -normal form (e.g., $(\lambda x. x x)(\lambda x. x x)$). Of course, if we restricted the input to simply-typed λ -terms, then the transformation would always terminate because the corresponding normalization function does.

We can modify the new CPS transformation to contract only the administrative redexes. We modify the datatype of intermediate terms (and the associated catamorphism operator) to contain two types of applications, corresponding to source and administrative redexes. This is an example of a general technique of embedding information known to the first pass in the structure of the intermediate language, for use by the second pass.

$$Term \ni m ::= \text{var } x \mid \text{lam}(x, m) \mid \text{app}(m, m) \mid \text{srcapp}(m, m)$$

We then modify the non-optimizing CPS transformation to preserve source applications (by replacing the application $\text{app}(\text{var } v_0, \text{var } v_1)$ with $\text{srcapp}(\text{var } v_0, \text{var } v_1)$ in the clause for applications); and we modify the normalization function (to always reify both the operator and operand of source applications). The datatype of normal forms now includes source redexes:

$$\begin{aligned} \text{Norm} \ni n &::= \text{atom}_N a \mid \text{lam}_N(x, n) \\ \text{Atom} \ni a &::= \text{var}_N x \mid \text{app}_N(a, n) \mid \text{srcapp}_N(n, n) \end{aligned}$$

The result of fusing the modified call-by-value CPS transformation with the modified normalization function is shown in Figure 6. Again, the domains of values and environments, and the auxiliary functions \downarrow and app are the same as in Figure 3.

$$\begin{aligned} \text{xform} & && : \text{Term} \rightarrow \text{Env} \rightarrow \text{Val} \\ \text{xform}(\text{var } x) \rho & && = \text{fun}(\lambda k. \text{app } k(\rho x)) \\ \text{xform}(\text{lam}(x, m)) \rho & && = \text{fun}(\lambda k. \text{app } k(\text{fun}(\lambda v. \text{xform } m(\rho\{x \mapsto v\})))) \\ \text{xform}(\text{app}(m_0, m_1)) \rho & && = \text{fun}(\lambda k. \text{app}(\text{xform } m_0 \rho) \\ & && \quad (\text{fun}(\lambda v_0. \text{app}(\text{xform } m_1 \rho) \\ & && \quad \quad (\text{fun}(\lambda v_1. \text{app}(\text{res}(\text{srcapp}_N(\downarrow v_0, \downarrow v_1))) k)))))) \\ \text{transform} & && : \text{Term} \rightarrow \text{Norm} \\ \text{transform } m & && = \downarrow(\text{xform } m \rho_{\text{init}}) \end{aligned}$$

FIGURE 6. A call-by-value CPS transformation that does not contract source redexes

7 COMPARISON TO DANVY AND FILINSKI'S ONE-PASS CPS TRANSFORMATION

Danvy and Filinski [11] obtained a one-pass CPS transformation by anticipating which administrative redexes would be built and contracting them at transformation time. They introduced a binding-time separation between static and dynamic constructs in the CPS transformation (static constructs are represented here by metalanguage variables, abstractions, and applications; and dynamic constructs by the constructors var , lam , and app). Static β -redexes are contracted at transformation time and dynamic redexes are residualized. We present their transformation in Figure 7.

In our transformation, the binding-time separation is present as well. Residualized atoms are dynamic and functions from values to values are static. This distinction arises naturally as a consequence of the residualizing model of the normalization function. Dynamic abstractions are only constructed by the auxiliary function \downarrow , and dynamic applications are only constructed by app .

Both CPS transformations are properly tail recursive: they do not generate η -redexes as the continuations of tail calls. In order to avoid generating this η -redex, Danvy and

$$\begin{aligned}
xform & : Term \rightarrow (Term \rightarrow Term) \rightarrow Term \\
xform (\text{var } x) & = \lambda \kappa. \kappa (\text{var } x) \\
xform (\text{lam } (x, m)) & = \lambda \kappa. \kappa (\text{lam } (x, \text{lam } (k, xform' m (\text{var } k)))) \\
xform (\text{app } (m_0, m_1)) & = \lambda \kappa. xform m_0 \\
& \quad (\lambda v_0. xform m_1 \\
& \quad \quad (\lambda v_1. \text{app } (\text{app } (\text{var } v_0, \text{var } v_1), \text{lam } (x, \kappa (\text{var } x))))) \\
\\
xform' & : Term \rightarrow Term \rightarrow Term \\
xform' (\text{var } x) & = \lambda k. \text{app } (k, \text{var } x) \\
xform' (\text{lam } (x, m)) & = \lambda k. \text{app } (k, \text{lam } (x, \text{lam } (k', xform' m (\text{var } k')))) \\
xform' (\text{app } (m_0, m_1)) & = \lambda k. xform m_0 \\
& \quad (\lambda v_0. xform m_1 \\
& \quad \quad (\lambda v_1. \text{app } (\text{app } (\text{var } v_0, \text{var } v_1), k))) \\
\\
transform & : Term \rightarrow Term \\
transform m & = \text{lam } (k, xform' m (\text{var } k))
\end{aligned}$$

FIGURE 7. Danvy and Filinski's one-pass CPS transformation

Filinski employ a pair of transformation functions, one for terms in tail position and one for terms in non-tail position. Our transformation uses a single transformation function for both terms in tail position and terms in non-tail position. The *app* function determines whether the operand of an application will be reified or not (reification will construct an η -expanded term if its argument is not already a normal-form atom).

8 A NEW ONE-PASS CALL-BY-NAME CPS TRANSFORMATION

The same fusion technique can be used with the CPS transformations for other evaluation orders [19]. For instance, we can start with Plotkin's call-by-name CPS transformation [22] shown in Figure 8.

$$\begin{aligned}
transform & : Term \rightarrow Term \\
transform (\text{var } x) & = \text{var } x \\
transform (\text{lam } (x, m)) & = \text{lam } (k, \text{app } (\text{var } k, \text{lam } (x, transform m))) \\
transform (\text{app } (m_0, m_1)) & = \text{lam } (k, \text{app } (transform m_0, \\
& \quad \text{lam } (v, \text{app } (\text{app } (\text{var } v, transform m_1), \text{var } k))))
\end{aligned}$$

FIGURE 8. Plotkin's non-optimizing call-by-name CPS transformation

After fusion and simplification, we obtain the one-pass call-by-name CPS transfor-

mation of Figure 9.

$$\begin{aligned}
xform & & : Term \rightarrow Env \rightarrow Val \\
xform (\text{var } x) \rho & = \rho x \\
xform (\text{lam } (x, m)) \rho & = \text{fun } (\lambda k. \text{app } k (\text{fun } (\lambda v. xform m (\rho \{x \mapsto v\})))) \\
xform (\text{app } (m_0, m_1)) \rho & = \text{fun } (\lambda k. \text{app } (xform m_0 \rho) \\
& \quad (\text{fun } (\lambda v. \text{app } (\text{app } v (xform m_1 \rho)) k))) \\
transform & & : Term \rightarrow Norm \\
transform m & = \downarrow (xform m \rho_{init})
\end{aligned}$$

FIGURE 9. A new one-pass call-by-name CPS transformation

The evaluation order of the normalization function is the same as that of the metalanguage. Due to the indifference theorems for both the call-by-value and call-by-name CPS transformations [22], the evaluation order of the normalization function is irrelevant here.

9 RELATED WORK AND CONCLUSION

This work brings together two strands of functional-programming research: program fusion and normalization by evaluation. It combines them to construct new one-pass CPS transformations based on NBE.

Program fusion. Techniques to eliminate intermediate data structures from functional programs have been an active area of research spanning three decades [6]. Wadler coined the term “deforestation” to describe the elimination of intermediate trees [26], and Gill et al. introduced the idea of using repeated application of the foldr/build rule for “shortcut” deforestation of intermediate lists [15]. Takano and Meijer extended shortcut deforestation to arbitrary polynomial datatypes [25]. Our contribution is the use of program-fusion techniques to construct one-pass transformations

Normalization by evaluation. The idea behind normalization by evaluation, that the metalanguage can be used to express normalization in the object language, is due to Martin L of [20]. This idea is present in Danvy and Filinski’s one-pass CPS transformation [10, 11], which is therefore an instance of NBE. Other examples include the free monoid [5], the untyped lambda-calculus and combinatory logic [16, 17, 18], the simply-typed λ -calculus [2, 4], and type-directed partial evaluation [8]. The term “normalization by evaluation” was coined by Schwichtenberg in 1998 [3]. Many people have discovered the same type-directed normalization function for the typed λ -calculus, using reify and reflect auxiliary functions [9]. The normalization function for the untyped λ -calculus has also been multiply discovered (e.g., by Coquand in the setting of dependent types [23]).

It has recently been investigated operationally by Aehlig and Joachimski [1] and denotationally by Filinski and Rohde [14]. Our contribution is to factor Danvy and Filinski's early example of NBE—the one-pass CPS transformation—into Plotkin's original CPS transformation and the normalization function for the untyped λ -calculus. The factorization scales to other CPS transformations [19] and more generally to other transformations on the λ -calculus.

NBE and the CPS transformation. Two other works combine normalization by evaluation with the CPS transformation. Danvy uses type-directed partial evaluation to residualize values produced by a continuation-passing evaluator for the λ -calculus [7], producing CPS terms in β -normal form; he does this for both call-by-value and call-by-name evaluators, yielding call-by-value and call-by-name CPS transformations. Filinski defines a (type-directed) extensional CPS transformation from direct-style values to CPS values and its inverse [13]; he composes this extensional CPS transformation with a type-directed reification function for the typed λ -calculus to obtain a transformation from direct-style values to CPS terms. We are not aware, however, of any other work combining the CPS transformation and reduction-free normalization using program fusion.

Acknowledgements. I wish to thank Olivier Danvy for his encouragement, his helpful discussions regarding normalization by evaluation, and for his comments.

REFERENCES

- [1] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14:587–611, 2004.
- [2] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.
- [3] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.
- [4] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
- [5] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.
- [6] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
- [7] Olivier Danvy. Décompilation de lambda-interprètes. In Guy Lapalme and Christian Queinsec, editors, *JFLA 96 – Journées francophones des langages applicatifs*, volume 15 of *Collection Didactique*, pages 133–146, Val-Morin, Québec, January 1996. INRIA.

- [8] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
- [9] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, University of Aarhus.
- [10] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
- [11] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
- [12] Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.
- [13] Andrzej Filinski. An extensional CPS transform (preliminary report). In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, Technical report 545, Computer Science Department, Indiana University, pages 41–46, London, England, January 2001.
- [14] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, number 2987 in Lecture Notes in Computer Science, pages 167–181, Barcelona, Spain, April 2002. Springer-Verlag.
- [15] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.
- [16] Mayer Goldberg. Gödelization in the λ -calculus. Technical Report BRICS RS-96-5, Computer Science Department, Aarhus University, Aarhus, Denmark, March 1996.
- [17] Mayer Goldberg. *Recursive Application Survival in the λ -Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.
- [18] Mayer Goldberg. Gödelization in the λ -calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.
- [19] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.
- [20] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.
- [21] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.
- [22] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [23] Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, Uppsala, Sweden, August 2003. ACM Press.

- [24] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, pages 124–132, 2002.
- [25] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 306–313, 1995.
- [26] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.