BRICS RS-07-6 Danvy et al.: On One-Pass CPS Transformations

BRICS

Basic Research in Computer Science

On One-Pass CPS Transformations

Olivier Danvy
Kevin Millikin
Lasse R. Nielsen

BRICS Report Series **RS-07-6**

ISSN 0909-0878 **March 2007**

**See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/07/6/`

On One-Pass CPS Transformations*

Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen
BRICS
Department of Computer Science
University of Aarhus†

March 21, 2007

Abstract

We bridge two distinct approaches to one-pass CPS transformations, i.e., CPS transformations that reduce administrative redexes at transformation time instead of in a post-processing phase. One approach is compositional and higher-order, and is independently due to Appel, Danvy and Filinski, and Wand, building on Plotkin's seminal work. The other is non-compositional and based on a reduction semantics for the λ-calculus, and is due to Sabry and Felleisen. To relate the two approaches, we use three tools: Reynolds's defunctionalization and its left inverse, refunctionalization; a special case of fold-unfold fusion due to Ohori and Sasano, fixed-point promotion; and an implementation technique for reduction semantics due to Danvy and Nielsen, refocusing.

This work is directly applicable to transforming programs into monadic normal form.

---

*Revised version of BRICS RS-02-3.
Theoretical Pearl to appear in the Journal of Functional Programming.
†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `<danvy@brics.dk>`, `<kmillikin@brics.dk>`, `<lrn@brics.dk>`

Contents

1 Introduction

Transforming functional programs into continuation-passing style (CPS) is a classical topic, with a long publication history [3,5,9,11–13,17,18,20,22,24–26,28,29,31,34–36,39–45,47,51,54–56,58–62,64–69,71,73],[1] including chapters in programming-languages textbooks [2,32,53], and many applications. Yet no standard CPS-transformation algorithm has emerged, and this missing piece contributes to maintaining continuations, CPS, and CPS transformations as mystifying artifacts (i.e., man-made constructs) in the land of programming and programming languages.

In this article, we bridge the two methodologically distinct CPS transformations described in the textbooks mentioned above. The first one, presented by Appel [2] and by Queinnec [53], is higher-order, and proceeds by recursive descent over the source program in a compositional way. The other one, presented by Friedman, Wand, and Haynes [32], is context-based, and rewrites the source program incrementally in a non-compositional way. Both transformations yield compact results, i.e., CPS programs without administrative redexes [17, 51, 60, 66]. The transformations reduce administrative redexes at transformation time and thus operate in one pass.

In the following sections, we inter-derive the higher-order transformation and the context-based transformation. The higher-order transformation is inspired by denotational semantics. It is compositional and uses a functional accumulator. The context-based transformation is inspired by reduction semantics, a variant of Plotkin's structural operational semantics [52] introduced in Felleisen's PhD thesis [26] and based on the notion of reduction contexts.

In a reduction semantics with applicative order for the λ-calculus, one defines terms, values, potential redexes, and contexts as follows:

$$
\begin{array}{rll}
x, k, w \in \mathit{Variables} & & \\
t \in \mathit{Terms} & t ::= v \mid t\, t \\
v \in \mathit{Values} & v ::= x \mid \lambda x.t \\
r \in \mathit{PotRedexes} & r ::= v\, v \\
C \in \mathit{Contexts} & C ::= [\,] \mid C[v\,[\,]] \mid C[[\,]\,t]
\end{array}
$$

In this semantics, the *unique-decomposition property* holds, i.e., any non-value term can be uniquely decomposed into a context and a potential redex (here: the application of a value to another value). One can therefore define a total function $\mathcal{D}$ that maps a value term to itself and a non-value term to a decomposition. There are many ways to define the $\mathcal{D}$ function, which is usually not shown in the literature. We use the following one here:

$$
\begin{array}{rl}
\mathcal{D} : \mathit{Terms} \rightarrow & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathcal{D}\, t = & \mathcal{D}'(t,\, [\,])
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{D}' : \mathit{Terms} \times \mathit{Contexts} \rightarrow & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathcal{D}'(v,\, C) = & \mathcal{D}'_{aux}(C,\, v) \\
\mathcal{D}'(t_0\, t_1,\, C) = & \mathcal{D}'(t_0,\, C[[\,]\, t_1])
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{D}'_{aux} : \mathit{Contexts} \times \mathit{Values} \rightarrow & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathcal{D}'_{aux}([\,],\, v) = & v \\
\mathcal{D}'_{aux}(C[[\,]\, t_1],\, v_0) = & \mathcal{D}'(t_1,\, C[v_0\,[\,]]) \\
\mathcal{D}'_{aux}(C[v_0\,[\,]],\, v_1) = & (C, v_0\, v_1)
\end{array}
$$

[1] Among many others.

This definition uses two auxiliary functions that are defined over the structure of their first argument: $\mathcal{D}'$ accumulates the spine context of an application, and $\mathcal{D}'_{aux}$ dispatches over the top constructor of the context. $\mathcal{D}'_{aux}$ could easily be inlined, giving

$$\begin{aligned}
\mathcal{D}'(\mathsf{v},\ [\,]) &= \mathsf{v} \\
\mathcal{D}'(\mathsf{v}_0,\ \mathsf{C}[[\,]\ \mathsf{t}_1]) &= \mathcal{D}'(\mathsf{t}_1,\ \mathsf{C}[\mathsf{v}_0\ [\,]]) \\
\mathcal{D}'(\mathsf{v}_1,\ \mathsf{C}[\mathsf{v}_0\ [\,]]) &= (\mathsf{C}, \mathsf{v}_0\ \mathsf{v}_1) \\
\mathcal{D}'(\mathsf{t}_0\ \mathsf{t}_1,\ \mathsf{C}) &= \mathcal{D}'(\mathsf{t}_0,\ \mathsf{C}[[\,]\ \mathsf{t}_1])
\end{aligned}$$

but we prefer to keep it since as stated above, this definition is in defunctionalized form [14]: the contexts form the data type of a defunctionalized function and $\mathcal{D}'_{aux}$ forms its apply function [19, 55]. We will exploit this property in Section 2.2.

Conversely, one can also define a total function $\mathcal{P}$ that plugs a term into a context (or again, as occasionally worded in the literature, that fills the hole of a context with a term, yielding another term). This $\mathcal{P}$ function is straighforwardly defined by structural induction over its first argument:

$$\begin{aligned}
\mathcal{P} &: \mathit{Contexts} \times \mathit{Terms} \to \mathit{Terms} \\
\mathcal{P}([\,],\ \mathsf{t}) &= \mathsf{t} \\
\mathcal{P}(\mathsf{C}[\mathsf{v}_0\ [\,]],\ \mathsf{t}_1) &= \mathcal{P}(\mathsf{C},\ \mathsf{v}_0\ \mathsf{t}_1) \\
\mathcal{P}(\mathsf{C}[[\,]\ \mathsf{t}_1],\ \mathsf{t}_0) &= \mathcal{P}(\mathsf{C},\ \mathsf{t}_0\ \mathsf{t}_1)
\end{aligned}$$

In essence, and as envisioned by Sabry and Felleisen [60], the context-based CPS transformation decomposes a source term into a context and a potential redex, CPS transforms the potential redex, plugs a fresh variable into the context, and iterates. It forms our starting point in Section 2.1. We then massage this context-based transformation until we obtain the usual higher-order one-pass CPS transformation. In Section 2.2, we start from this higher-order one-pass CPS transformation and we walk back to the context-based CPS transformation.

The rest of the article builds on Section 2. In Section 3, we refine the CPS transformation to make it tail-conscious, to avoid spurious administrative η-redexes in the CPS counterpart of source tail calls. Section 4 compares and contrasts the two standard variants of continuation-passing style, i.e., with continuations first or last. We review the administrative η-reductions enabled by each variant. Section 5 addresses generalized reduction and how to integrate it in both the context-based and the higher-order one-pass CPS transformations. Finally, in Section 6, we put everything together and assemble a tail-conscious CPS transformation with administrative η-reductions and that integrates generalized reduction. The continuations-first variant of the result is the CPS transformation designed by Sabry and Felleisen for reasoning about programs in continuation-passing style [60].

Prerequisites: We assume a basic familiarity with the λ-calculus [4], with reduction semantics [21, 26, 27, 72], and with the notion of one-pass CPS transformation [17, 60]. We also make use of Reynolds's defunctionalization, i.e., the data-structure representation of higher-order functions [19, 55] and of its left inverse, refunctionalization [15].

2 Standard CPS transformation

2.1 From context-based to higher-order

The following left-to-right, call-by-value CPS transformation repeatedly decomposes a source term into a context and the application of a pair of values, CPS transforms the application, and plugs a fresh variable into the context. This process continues until the source term is a value.

Definition 1 (Implicit context-based CPS transformation)

$$
\begin{aligned}
\mathcal{T} : \mathit{Terms} \times \mathit{Variables} &\to \mathit{Terms} \\
\mathcal{T}\,(v,\, k) &= k\,(\mathcal{V}\,v) \\
\mathcal{T}\,(C[v_0\,v_1],\, k) &= (\mathcal{V}\,v_0)\,(\mathcal{V}\,v_1)\,(\mathcal{C}\,(C,\, k))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V} : \mathit{Values} &\to \mathit{Values} \\
\mathcal{V}\,x &= x \\
\mathcal{V}\,\lambda x.t &= \lambda x.\lambda k.\mathcal{T}\,(t,\, k) \\
&\quad \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C} : \mathit{Contexts} \times \mathit{Variables} &\to \mathit{Values} \\
\mathcal{C}\,(C,\, k) &= \lambda w.\mathcal{T}\,(C[w],\, k) \\
&\quad \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\lambda k.\mathcal{T}\,(t,\, k)$, *where* k *is fresh.* □

Implicit in Definition 1 are the *decomposition* of a non-value source expression into a context and a potential redex (on the left-hand side of the second clause of the definition of $\mathcal{T}$) and the *plugging* of an expression into a context (on the right-hand side of the definition of $\mathcal{C}$). Here is an explicit version of this definition, using $\mathcal{D}$ and $\mathcal{P}$ as defined in Section 1:

Definition 2 (Explicit context-based CPS transformation)

$$
\begin{aligned}
\mathcal{T} : (\mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes}) \times \mathit{Variables} &\to \mathit{Terms} \\
\mathcal{T}\,(v,\, k) &= k\,(\mathcal{V}\,v) \\
\mathcal{T}\,((C, v_0\,v_1),\, k) &= (\mathcal{V}\,v_0)\,(\mathcal{V}\,v_1)\,(\mathcal{C}\,(C,\, k))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V} : \mathit{Values} &\to \mathit{Values} \\
\mathcal{V}\,x &= x \\
\mathcal{V}\,\lambda x.t &= \lambda x.\lambda k.\mathcal{T}\,(\mathcal{D}\,t,\, k) \\
&\quad \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C} : \mathit{Contexts} \times \mathit{Variables} &\to \mathit{Values} \\
\mathcal{C}\,(C,\, k) &= \lambda w.\mathcal{T}\,(\mathcal{D}\,(\mathcal{P}(C,\, w)),\, k) \\
&\quad \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\lambda k.\mathcal{T}\,(\mathcal{D}\,t,\, k)$, *where* k *is fresh.* □

If they are implemented literally, decomposition and plugging entail a time factor that is linear in the size of the source program, in the worst case. Overall, the worst-case time complexity of the CPS transformation is then quadratic in the size of the source program [21], which is an overkill since $\mathcal{D}$ is always applied to the result of $\mathcal{P}$. (We write 'always' since $\mathcal{D}\,t = \mathcal{D}\,(\mathcal{P}([\,],\, t))$.)

Danvy and Nielsen [21,48] have shown that the composition of plugging and decomposition can be fused into a 'refocus' function $\mathcal{R}$ that makes the resulting CPS transformation operate in time linear in the size of the source program—or more precisely, in one pass. The essence of refocusing for a reduction semantics satisfying the unique decomposition property is captured in the following proposition:

Proposition 1 (Danvy & Nielsen [14, 21]) *For any term* t *and context* C, $\mathcal{D}\,(\mathcal{P}(C,\ t)) = \mathcal{D}'(t,\ C)$.
□

In words: refocusing amounts to continuing the decomposition of the given term in the given context. Intuitively, $\mathcal{R}$ maps a term and a context into the next context and potential redex, if there is any.

The definition of $\mathcal{R}$ is therefore a clone of that of $\mathcal{D}'$. In particular, it involves an auxiliary function $\mathcal{R}'$ and takes the form of two state-transition functions:

$$
\begin{aligned}
\mathcal{R} : \mathit{Terms} \times \mathit{Contexts} &\rightarrow \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathcal{R}(v,\ C) &= \mathcal{R}'(C,\ v) \\
\mathcal{R}(t_0\, t_1,\ C) &= \mathcal{R}(t_0,\ C[[\,]\, t_1])
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}' : \mathit{Contexts} \times \mathit{Values} &\rightarrow \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathcal{R}'([\,],\ v) &= v \\
\mathcal{R}'(C[[\,]\, t_1],\ v_0) &= \mathcal{R}(t_1,\ C[v_0\, [\,]]) \\
\mathcal{R}'(C[v_0\, [\,]],\ v_1) &= (C, v_0\, v_1)
\end{aligned}
$$

(Again, $\mathcal{R}'$ could be inlined.)

We take this one-pass CPS transformation as the starting point of our derivation:

Definition 3 (Context-based CPS transformation, refocused)

$$
\begin{aligned}
\mathcal{T}_1 : (\mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes}) \times \mathit{Variables} &\rightarrow \mathit{Terms} \\
\mathcal{T}_1\,(v,\ k) &= k\,(\mathcal{V}_1\, v) \\
\mathcal{T}_1\,((C, v_0\, v_1),\ k) &= (\mathcal{V}_1\, v_0)\,(\mathcal{V}_1\, v_1)\,(\mathcal{C}_1\,(C,\ k))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_1 : \mathit{Values} &\rightarrow \mathit{Values} \\
\mathcal{V}_1\, x &= x \\
\mathcal{V}_1\, \lambda x.t &= \lambda x.\lambda k.\mathcal{T}_1\,(\mathcal{R}(t,\ [\,]),\ k) \\
&\ \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_1 : \mathit{Contexts} \times \mathit{Variables} &\rightarrow \mathit{Values} \\
\mathcal{C}_1\,(C,\ k) &= \lambda w.\mathcal{T}_1\,(\mathcal{R}(w,\ C),\ k) \\
&\ \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\lambda k.\mathcal{T}_1\,(\mathcal{R}(t,\ [\,]),\ k)$, *where* k *is fresh.* □

In Definition 2, $\mathcal{D}$ was always applied to the result of $\mathcal{P}$. Similarly, in Definition 3, $\mathcal{T}_1$ is always applied to the result of $\mathcal{R}$. Ohori and Sasano [49] have shown that the composition of functions such as $\mathcal{T}_1$ and $\mathcal{R}$ can be fused by 'fixed-point promotion' into a function $\mathcal{R}_{\mathcal{T}_2}$ in such a way that for any term t, context C, and continuation identifier k,

$$\mathcal{T}_1\,(\mathcal{R}(t,\ C),\ k) = \mathcal{R}_{\mathcal{T}_2}(t,\ C,\ k).$$

We detail this fusion in Appendices A and B. The resulting fused CPS transformation reads as follows:

Definition 4 (Context-based CPS transformation, fused)

$$
\begin{aligned}
\mathcal{R}_{\mathcal{T}_2} &: \textit{Terms} \times \textit{Contexts} \times \textit{Variables} \rightarrow \textit{Terms} \\
\mathcal{R}_{\mathcal{T}_2}(v,\ C,\ k) &= \mathcal{R}'_{\mathcal{T}_2}(C,\ v,\ k) \\
\mathcal{R}_{\mathcal{T}_2}(t_0\, t_1,\ C,\ k) &= \mathcal{R}_{\mathcal{T}_2}(t_0,\ C[[\,]\, t_1],\ k)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R}'_{\mathcal{T}_2} &: \textit{Contexts} \times \textit{Values} \times \textit{Variables} \rightarrow \textit{Terms} \\
\mathcal{R}'_{\mathcal{T}_2}([\,],\ v,\ k) &= k\,(\mathcal{V}_2\, v) \\
\mathcal{R}'_{\mathcal{T}_2}(C[[\,]\, t_1],\ v_0,\ k) &= \mathcal{R}_{\mathcal{T}_2}(t_1,\ C[v_0\, [\,]],\ k) \\
\mathcal{R}'_{\mathcal{T}_2}(C[v_0\, [\,]],\ v_1,\ k) &= (\mathcal{V}_2\, v_0)\ (\mathcal{V}_2\, v_1)\ (\mathcal{C}_2\,(C,\ k))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_2 &: \textit{Values} \rightarrow \textit{Values} \\
\mathcal{V}_2\, x &= x \\
\mathcal{V}_2\, \lambda x.t &= \lambda x.\lambda k.\mathcal{R}_{\mathcal{T}_2}(t,\ [\,],\ k) \\
&\ \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_2 &: \textit{Contexts} \times \textit{Variables} \rightarrow \textit{Values} \\
\mathcal{C}_2\,(C,\ k) &= \lambda w.\mathcal{R}_{\mathcal{T}_2}(w,\ C,\ k) \\
&\ \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\lambda k.\mathcal{R}_{\mathcal{T}_2}(t,\ [\,],\ k)$*, where* k *is fresh.* □

Because the contexts are solely consumed by the rules defining $\mathcal{R}'_{\mathcal{T}_2}$, this CPS transformation is in the image of Reynolds's defunctionalization. The contexts are a first-order representation of the function type *Values* × *Variables* → *Terms* with $\mathcal{R}'_{\mathcal{T}_2}$ as the apply function. As the last step of the derivation, let us therefore refunctionalize this CPS transformation.

Under the assumption that C is refunctionalized as $\widehat{C}$, and for any t and k, we define $\mathcal{R}_{\mathcal{T}_3}(\widehat{C},\ t,\ k)$ to equal $\mathcal{R}_{\mathcal{T}_2}(C,\ t,\ k)$, and we write $\mathcal{V}_3$ and $\mathcal{C}_3$ to denote the counterparts of $\mathcal{V}_2$ and $\mathcal{C}_2$ over refunctionalized contexts. We introduce the infix operator @ for applications, and we overline λ and @ for the static abstractions and applications introduced by refunctionalization; we also write u for the corresponding static variables. Symmetrically, we underline λ and @ for the dynamic abstractions and applications constructing the residual CPS program, and we write w for the corresponding dynamic variables.

- $[\,]$ is refunctionalized as
$$\overline{\lambda} u.\overline{\lambda} k.k \,\underline{@}\, (\mathcal{V}_3\, u),$$
corresponding to the first rule of $\mathcal{R}'_{\mathcal{T}_2}$;

- if C is refunctionalized as $\widehat{C}$ then $C[v_0\, [\,]]$ is refunctionalized as
$$\overline{\lambda} u_1.\overline{\lambda} k.(\mathcal{V}_3\, v_0) \,\underline{@}\, (\mathcal{V}_3\, u_1) \,\underline{@}\, (\mathcal{C}_3(\widehat{C},\ k)),$$
corresponding to the third rule of $\mathcal{R}'_{\mathcal{T}_2}$; and

- if C is refunctionalized as $\widehat{C}$ then $C[[\,]\, t_1]$ is refunctionalized as
$$\overline{\lambda} u_0.\overline{\lambda} k.\mathcal{R}_{\mathcal{T}_3}(t_1,\ \overline{\lambda} u_1.\overline{\lambda} k.(\mathcal{V}_3\, u_0) \,\underline{@}\, (\mathcal{V}_3\, u_1) \,\underline{@}\, (\mathcal{C}_3(\widehat{C},\ k)),\ k),$$
corresponding to the second rule of $\mathcal{R}'_{\mathcal{T}_2}$.

The interpretation of contexts previously performed by $\mathcal{R}'_{\mathcal{T}_2}$ is now performed by static application.

An improvement: instead of $\mathcal{R}_{\mathcal{T}_3}$ that operates on t, $\widehat{C}$, and k, we can instead apply the refunctionalized context $\widehat{C}$ to the continuation identifier k as soon as it is available. To this end, we define a function $\mathcal{T}_3$ operating on t and on $\overline{\lambda}u.\widehat{C}\,\overline{@}\,u\,\overline{@}\,k$, so that $\mathcal{R}_{\mathcal{T}_3}(t,\ \widehat{C},\ k) = \mathcal{T}_3(t,\ \overline{\lambda}u.\widehat{C}\,\overline{@}\,u\,\overline{@}\,k)$. The result is the following higher-order CPS transformation:

Definition 5 (Context-based CPS transformation, refunctionalized)

$$\begin{aligned}
\mathcal{T}_3 : \mathit{Terms} \times (\mathit{Values} \to \mathit{Terms}) &\to \mathit{Terms} \\
\mathcal{T}_3(v,\ \kappa) &= \kappa\,\overline{@}\,v \\
\mathcal{T}_3(t_0\, t_1,\ \kappa) &= \mathcal{T}_3(t_0,\ \overline{\lambda}u_0.\mathcal{T}_3(t_1,\ \overline{\lambda}u_1.(\mathcal{V}_3\, u_0)\,\underline{@}\,(\mathcal{V}_3\, u_1)\,\underline{@}\,(\mathcal{C}_3\,\kappa)))
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}_3 : \mathit{Values} &\to \mathit{Values} \\
\mathcal{V}_3\, x &= x \\
\mathcal{V}_3\, \lambda x.t &= \underline{\lambda} x.\underline{\lambda} k.\mathcal{T}_3(t,\ \overline{\lambda}u.k\,\underline{@}\,(\mathcal{V}_3\, u)) \\
&\quad \textit{where } k \textit{ is fresh}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_3 : (\mathit{Values} \to \mathit{Terms}) &\to \mathit{Values} \\
\mathcal{C}_3\, \kappa &= \underline{\lambda} w.\kappa\,\overline{@}\,w \\
&\quad \textit{where } w \textit{ is fresh}
\end{aligned}$$

The CPS transformation of a program t *is* $\underline{\lambda}k.\mathcal{T}_3(t,\ \overline{\lambda}u.k\,\underline{@}\,(\mathcal{V}_3\, u))$, *where* k *is fresh.* □

This CPS transformation is very close to the usual higher-order one-pass CPS transformation. It is manifestly not compositional, witness the applications of $\mathcal{V}_3$ to the static variables u_0, u_1, and u. This non-compositionality is directly inherited from the initial context-based CPS transformation, which is also non-compositional.

The non-compositionality can be read off the types if we write *DTerms* and *DValues* for the syntactic domains of source direct-style expressions and values and *CTerms* and *CValues* for the syntactic domains of target CPS expressions and values. The types of $\mathcal{T}_3$, $\mathcal{V}_3$, and $\mathcal{C}_3$ are then as follows:

$$\begin{aligned}
\mathcal{T}_3 &: \mathit{DTerms} \to (\mathit{DValues} \to \mathit{CTerms}) \to \mathit{CTerms} \\
\mathcal{V}_3 &: \mathit{DValues} \to \mathit{CValues} \\
\mathcal{C}_3 &: (\mathit{DValues} \to \mathit{CTerms}) \to \mathit{CValues}
\end{aligned}$$

We can easily make this CPS transformation compositional by applying $\mathcal{V}$ prior to applying κ instead of afterwards. The types of the resulting compositional functions $\mathcal{T}_4$ and $\mathcal{C}_4$ then read as follows:

$$\begin{aligned}
\mathcal{T}_4 &: \mathit{DTerms} \to (\mathit{CValues} \to \mathit{CTerms}) \to \mathit{CTerms} \\
\mathcal{C}_4 &: (\mathit{CValues} \to \mathit{CTerms}) \to \mathit{CValues}
\end{aligned}$$

The result is then the usual higher-order one-pass CPS transformation, which is our starting point in Section 2.2.

2.2 From higher-order to context-based

Appel [2], Danvy and Filinski [16, 17], and Wand [71] each discovered the following higher-order one-pass CPS transformation:

Definition 6 (Higher-order CPS transformation)

$$
\begin{aligned}
\mathcal{T}_4 : \mathit{DTerms} \times (\mathit{CValues} \rightarrow \mathit{CTerms}) &\rightarrow \mathit{CTerms} \\
\mathcal{T}_4(v,\ \kappa) &= \kappa\,\overline{@}\,\mathcal{V}_4\,v \\
\mathcal{T}_4(t_0\,t_1,\ \kappa) &= \mathcal{T}_4(t_0,\ \overline{\lambda} u_0.\mathcal{T}_4(t_1,\ \overline{\lambda} u_1.u_0\,\underline{@}\,u_1\,\underline{@}\,(\mathcal{C}_4\,\kappa)))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_4 : \mathit{DValues} &\rightarrow \mathit{CValues} \\
\mathcal{V}_4\,x &= x \\
\mathcal{V}_4\,\lambda x.t &= \underline{\lambda} x.\underline{\lambda} k.\mathcal{T}_4(t,\ \overline{\lambda} u.k\,\underline{@}\,u) \\
& \quad \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_4 : (\mathit{CValues} \rightarrow \mathit{CTerms}) &\rightarrow \mathit{CValues} \\
\mathcal{C}_4\,\kappa &= \underline{\lambda} w.\kappa\,\overline{@}\,w \\
& \quad \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\underline{\lambda} k.\mathcal{T}_4(t,\ \overline{\lambda} u.k\,\underline{@}\,u)$*, where* k *is fresh.* □

Let us defunctionalize this higher-order transformation. The type *CValues* $\rightarrow$ *CTerms* is inhabited by instances of three λ-abstractions (the overlined ones in Definition 6). It therefore gives rise to a data type with three constructors (written below as in ML) and its associated apply function interpreting these constructors.

The corresponding defunctionalized CPS transformation reads as follows:

Definition 7 (Higher-order CPS transformation, defunctionalized)

$$
\begin{aligned}
\textit{datatype Fun} &= F_0 \textit{ of } \textit{Variables} \\
&\mid F_1 \textit{ of } \textit{Fun} \times \textit{DTerms} \\
&\mid F_2 \textit{ of } \textit{Fun} \times \textit{CValues}
\end{aligned}
$$

$$
\begin{aligned}
\textit{apply}_5 : \textit{Fun} \times \textit{CValues} &\rightarrow \textit{CTerms} \\
\textit{apply}_5(F_0\,k,\ u) &= k\,\underline{@}\,u \\
\textit{apply}_5(F_1\,(f,\ t_1),\ u_0) &= \mathcal{T}_5(t_1,\ F_2\,(f,\ u_0)) \\
\textit{apply}_5(F_2\,(f,\ u_0),\ u_1) &= u_0\,\underline{@}\,u_1\,\underline{@}\,(\mathcal{C}_5\,f)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}_5 : \textit{DTerms} \rightarrow \textit{Fun} &\rightarrow \textit{CTerms} \\
\mathcal{T}_5(v,\ f) &= \textit{apply}_5(f,\ \mathcal{V}_5\,v) \\
\mathcal{T}_5(t_0\,t_1,\ f) &= \mathcal{T}_5(t_0,\ F_1\,(f,\ t_1))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_5 : \textit{DValues} &\rightarrow \textit{CValues} \\
\mathcal{V}_5\,x &= x \\
\mathcal{V}_5\,\lambda x.t &= \underline{\lambda} x.\underline{\lambda} k.\mathcal{T}_5(t,\ F_0\,k) \\
& \quad \textit{where } k \textit{ is fresh}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_5 : \textit{Fun} &\rightarrow \textit{CValues} \\
\mathcal{C}_5\,f &= \underline{\lambda} w.\textit{apply}_5(f,\ w) \\
& \quad \textit{where } w \textit{ is fresh}
\end{aligned}
$$

The CPS transformation of a program t *is* $\underline{\lambda} k.\mathcal{T}_5(t,\ F_0\,k)$*, where* k *is fresh.* □

We recognize the result as a refocused context-based CPS transformation where the contexts hold elements of *CValues* instead of elements of *DValues*. The data type *Fun* plays the role of the contexts (indexing each empty context with a continuation identifier), $apply_5$ plays the role of $\mathcal{R}'_{\mathcal{T}_2}$, and $\mathcal{T}_5$ plays the role of $\mathcal{R}_{\mathcal{T}_2}$.

Alternatively, we can defunctionalize the CPS transformation of Definition 6 so that the data type and the type of its apply function read as follows:[2]

$$\begin{aligned} \mathit{datatype\ Fun} = {} & \mathsf{F}_0 \ \mathit{of\ Variables} \\ \mid {} & \mathsf{F}_1 \ \mathit{of\ Fun} \times \mathit{DTerms} \\ \mid {} & \mathsf{F}_2 \ \mathit{of\ Fun} \times \mathit{DValues} \end{aligned}$$

$$\mathit{apply} : \mathit{Fun} \times \mathit{DValues} \to \mathit{CTerms}$$

We then obtain the CPS transformation of Definition 4.

2.3 Summary and conclusion

We have bridged two approaches to one-pass CPS transformations, one that is context-based and non-compositional, and the other that is higher-order and compositional. This bridge is significant because even though they share the same goal, the two approaches have been developed independently and have always been reported separately in the literature.

We have used three tools to bridge the two CPS transformations: refocusing, fixed-point promotion, and defunctionalization. Refocusing short-cuts plugging and decomposition, and made it possible for the context-based CPS transformation to operate in one pass. Fixed-point promotion is a special case of fold/unfold fusion, and made it possible to fuse the resulting CPS transformation with its refocus function.[3] Defunctionalization and its left inverse, refunctionalization, are changes of representation between the higher-order world and the first-order world, and they made it possible to relate higher-order and context-based CPS transformations.

3 Tail-conscious CPS transformation

The CPS transformations of Section 2 generate one η-redex for each source tail-call. For example, they map a term such as $\lambda x.f\ (g\ x)$ into the following one:

$$\lambda k.k\ (\lambda x.\lambda k.g\ x\ (\lambda w.f\ w\ (\lambda w'.k\ w')))$$

In this CPS term, the continuation of the (tail) call to f is $\lambda w'.k\ w'$.

In contrast, a tail-conscious CPS transformation would yield the following η-reduced term:

$$\lambda k.k\ (\lambda x.\lambda k.g\ x\ (\lambda w.f\ w\ k))$$

Tail-consciousness matters for readability and in CPS-based compilers.

3.1 Making a context-based CPS transformation tail-conscious

The specification of $\mathcal{C}$ in Definition 2 can be refined as follows to make it tail-conscious:

$$\begin{aligned} & \mathcal{C} : \mathit{Contexts} \times \mathit{Variables} \to \mathit{Values} \\ & \qquad \mathcal{C}\ ([\,],\ k) = k \\ & \qquad \mathcal{C}\ (C,\ k) = \lambda w.\mathcal{T}\ (C[w],\ k) \qquad \text{if } C \neq [\,] \\ & \qquad\qquad\qquad\quad \text{where } w \text{ is fresh} \end{aligned}$$

[2] This choice in defining a data type is similar to the choice between minimally free expressions and maximally free expressions in super-combinator conversion [50, pages 245–247].

[3] In another context [7,8,14,21], fixed-point promotion makes it possible to transform a 'pre-abstract machine' into a 'staged abstract machine'.

One can then take the same steps as in Section 2.1 to obtain a tail-conscious higher-order CPS transformation similar to Danvy and Filinski's [17].

3.2 Making a higher-order CPS transformation tail-conscious

The specification in Definition 6 can be refined to make it tail-conscious. The idea is to make the second parameter of $\mathcal{T}_4$ a sum, i.e., either the continuation identifier (in case of source tail call), or a function.

$$\mathcal{T}_4 : DTerms \times (Variables + CValues \to CTerms) \to CTerms$$
$$\mathcal{C}_4 : Variables + CValues \to CTerms \to CValues$$

(Alternatively, the definition of $\mathcal{T}_4$ can be split into two, one for each summand.) One can then take the same steps as in Section 2.2 to obtain a tail-conscious context-based CPS transformation similar to the one of Section 3.1.

4 Continuations first or continuations last?

When writing a continuation-passing λ-abstraction, should one write $\lambda x.\lambda k.t$ or $\lambda k.\lambda x.t$? Since Plotkin [51] and Steele [66], tradition has it to do the former, but the latter makes curried continuation-passing functions continuation transformers [1, 33]. Because this order was first promoted in Fischer's work [29],[4] putting continuations first is said to be "à la Fischer" and is used, e.g., by Fradet and Le Métayer [31], by Sabry and Felleisen [60], and by Reppy [54]. Conversely, putting continuations last is said to be "à la Plotkin" and is used more frequently.

Sections 2 and 3 are concerned with CPS à la Plotkin, but their content can be adapted mutatis mutandis to CPS à la Fischer. On the other hand, each flavor of CPS enables new and distinct opportunities for administrative η-reductions, which are a source of compactness in CPS programs.

Tail-conscious CPS à la Plotkin: In a λ-abstraction, a tail call where sub-terms are values such as in $\lambda y.f\ x$ is transformed into $\lambda k.k\ (\lambda y.\lambda k.f\ x\ k)$, where the inner continuation can be η-reduced.

Tail-conscious CPS à la Fischer: A term containing nested applications such as $\lambda x.f\ (g\ (h\ x))$ is transformed into $\lambda k.k\ (\lambda k.\lambda x.h\ (\lambda w_1.g\ (\lambda w_2.f\ k\ w_2)\ w_1)\ x)$. In this CPS term, the parameter of each continuation can be administratively η-reduced, producing the following term, where indeed even x can be η-reduced:

$$\lambda k.k\ (\lambda k.\lambda x.h\ (g\ (f\ k))\ x)$$

As the two examples illustrate, a curried CPS à la Plotkin makes it possible to η-reduce continuation identifiers for some source λ-abstractions, whereas a curried CPS à la Fischer makes it possible to η-reduce parameters of continuations for some source applications. Since, on the average, there are many more applications than abstractions in a λ-term, by construction, the Fischer curried flavor offers more opportunities than the Plotkin curried flavor for obtaining compact CPS programs through administrative η-reductions.

Furthermore, it is possible to perform administrative η-reductions at transformation time, i.e., in one pass. One is, however, left with the task of proving that administrative η-reductions

[4] On pragmatic grounds—using cons rather than append over lists of parameters in uncurried CPS.

are *value* η-reductions, i.e., that they do not alter the properties of CPS-transformed programs, namely simulation, indifference, and translation [38, 51] as well as termination.

At any rate, the current agreement in the continuation community is that administrative η-reductions bring more trouble than benefits. In fact, for uncurried CPS, neither flavor provides any extra opportunity for administrative η-reduction beyond tail consciousness. In short, only tail-consciousness matters, and it works both for Plotkin and Fischer, uniformly.

5 CPS transformation with generalized reduction

5.1 Generalized reduction

In his PhD thesis [58, 60], Sabry considered β_{lift}, a generalized reduction that is most easily described using reduction contexts [10]:

$$C[(\lambda x.t_0)\ t_1] \longrightarrow_{\beta_{lift}} (\lambda x.C[t_0])\ t_1$$

A β_{lift}-reduction in the direct-style world corresponds to an administrative (i.e., overlined) β-reduction in the corresponding CPS program à la Fischer:

$$((\overline{\lambda} k.\underline{\lambda} x.t_0')\ \overline{@}\ c)\ \underline{@}\ v_1' \longrightarrow_{adm} (\underline{\lambda} x.t_0'[c/k])\ \underline{@}\ v_1'$$

(t_0' is the CPS counterpart of t_0, v_1' is the CPS counterpart of t_1, and c represents C.)

Similarly, a β_{lift}-reduction in the direct-style world corresponds to an administrative generalized β-reduction in the corresponding CPS program à la Plotkin:

$$((\underline{\lambda} x.\overline{\lambda} k.t_0')\ \underline{@}\ v_1')\ \overline{@}\ c \longrightarrow_{adm} (\underline{\lambda} x.t_0'[c/k])\ \underline{@}\ v_1'$$

5.2 Administrative generalized reduction

Integrating β_{lift} into the CPS transformation is achieved by refining the following rule in Definition 2:

$$\mathcal{T}\ (C[v_0\ v_1],\ k) = (\mathcal{V}\ v_0)\ (\mathcal{V}\ v_1)\ (\mathcal{C}\ (C,\ k))$$

The idea is to enumerate the possible instances of v_0, i.e., whether it denotes a variable or a λ-abstraction:

$$\begin{aligned} \mathcal{T}\ (C[x\ v_1],\ k) &= x\ (\mathcal{V}\ v_1)\ (\mathcal{C}\ (C,\ k)) \\ \mathcal{T}\ (C[(\lambda x.t_0)\ v_1],\ k) &= (\lambda x.\mathcal{T}\ (C[t_0],\ k))\ (\mathcal{V}\ v_1) \\ & \qquad \text{renaming } x \text{ if it occurs free in } C \end{aligned}$$

As in Section 2, the refined context-based CPS transformation can be refocused to operate in one-pass and refunctionalized to be higher-order. Making it compositional, however, makes the CPS transformation dependently typed [22]. The steps are reversible, turning a one-pass higher-order CPS transformation with generalized reduction into a one-pass refocused context-based CPS transformation.

6 Tail-conscious CPS transformation à la Fischer with administrative η-reductions and generalized reduction

Putting everything together, Definition 2 can be made tail-conscious and extended with administrative η-reductions and generalized reduction. The result, if it is à la Fischer, coincides with Sabry and Felleisen's compacting CPS transformation (Sabry & Felleisen, 1993, Definition 5). It can be refocused to operate in one-pass and refunctionalized to be higher-order. But as in Section 5, making it compositional makes the CPS transformation dependently typed [22]. The derivation steps are reversible.

7 Conclusions and issues

We have connected two distinct approaches to a one-pass CPS transformation that have been reported separately in the literature. One is higher-order and compositional, stems from denotational semantics, and can be expressed directly as a functional program. The other is rewriting-based and non-compositional, stems from reduction semantics, and requires an adaptation such as refocusing to operate in one pass. The connection between the two approaches reduces their choice to a matter of convenience.

While all textbook descriptions of the one-pass CPS transformation [2, 32, 53] account for tail-consciousness, none pays a particular attention to administrative η-reductions and to generalized reduction. For example, the context-based CPS transformation of the second edition of *Essentials of Programming Languages* [32] produces uncurried CPS programs à la Plotkin and corresponds to the content of Section 3.

The derivation steps presented in the present article can be used for richer languages, i.e., languages with literals, primitive operations, conditional expressions, block structure, and computational effects (state, control, etc.). They also directly apply to transforming programs into monadic normal form [6, 30, 37, 46].

Acknowledgments: Thanks are due to Julia L. Lawall for comments and to an anonymous reviewer for pressing us to spell out how $\mathcal{T}_1$ and $\mathcal{R}$ are fused; Ohori and Sasano's fixed-point promotion provides a particularly concise explanation for this fusion.

This work is partly supported by the Danish Natural Science Research Council, Grant no. 21-03-0545.

A Fixed-point promotion

We outline Ohori and Sasano's fixed-point promotion algorithm [49] and illustrate it with a simple example.

Fixed-point promotion fuses the composition $f \circ g$ of a strict function f and a recursive function g. As a simple example, consider a function computing the run-length encoding of a list. Given a list of elements, this function segments it into a list of pairs of elements and non-negative integers, replacing each longest sequence s of consecutive identical elements x in the given list with a pair (x, n) where n is the length of s. For example, it maps $[W, W, B, B, B]$ into $[(W, 2), (B, 3)]$.

We make use of an auxiliary tail-recursive function *next* that traverses a segment and computes its length. Additionally, if the rest of the list is nonempty, it also returns its head and tail:

$$
\begin{array}{l}
\mathit{next} : \alpha \times \mathit{List}(\alpha) \times \mathit{Nat} \rightarrow \alpha \times \mathit{Nat} \times (\mathit{Unit} + \alpha \times \mathit{List}(\alpha)) \\
\qquad\qquad \begin{array}{rcl}
\mathit{next}\ (x, \mathit{nil}, n) & = & (x,\ n,\ ()) \\
\mathit{next}\ (x, x' :: \mathit{xs}, n) & = & \mathit{if}\ x = x'\ \mathit{then}\ \mathit{next}\ (x, \mathit{xs}, n+1)\ \mathit{else}\ (x,\ n,\ (x',\ \mathit{xs}))
\end{array}
\end{array}
$$

A second auxiliary function *continue* dispatches on the return value of *next* and continues encoding the tail of the list if necessary:

$$
\begin{array}{l}
\mathit{continue} : \alpha \times \mathit{Nat} \times (\mathit{Unit} + \alpha \times \mathit{List}(\alpha) \rightarrow \mathit{List}(\alpha \times \mathit{Nat}) \\
\qquad\qquad \begin{array}{rcl}
\mathit{continue}\ (x,\ n,\ ()) & = & (x,\ n) :: \mathit{nil} \\
\mathit{continue}\ (x,\ n,\ (x',\ \mathit{xs})) & = & (x,\ n) :: \mathit{continue}\ (\mathit{next}\ (x', \mathit{xs}, 1))
\end{array}
\end{array}
$$

The run-length encoding of a list is then the composition of *continue* and *next*:

$$\begin{aligned}
\mathit{encode} : \mathit{List}(\alpha) &\to \mathit{List}(\alpha \times \mathit{Nat}) \\
\mathit{encode}\ \mathit{nil} &= \mathit{nil} \\
\mathit{encode}\ (x :: \mathit{xs}) &= \mathit{continue}\ (\mathit{next}\ (x, \mathit{xs}, 1))
\end{aligned}$$

To fuse this composition, we will use fixed-point promotion and proceed accordingly in four steps.

The first step is to inline the application of *next* in the composition to expose its body to *continue*:

$$\begin{aligned}
& \lambda(x,\ \mathit{xs},\ n).\mathit{continue}\ (\mathit{next}\ (x, \mathit{xs}, n)) \\
= \quad & \{\mathit{inline\ next}\} \\
& \lambda(x,\ \mathit{xs},\ n).\mathit{continue}\ (\mathit{case}\ (x,\ \mathit{xs},\ n) \\
& \qquad \mathit{of}\ (x,\ \mathit{nil},\ n) \Rightarrow (x,\ n,\ ()) \\
& \qquad \mid (x,\ x' :: \mathit{xs},\ n) \Rightarrow\ \mathit{if}\ x = x' \\
& \qquad\qquad \mathit{then}\ \mathit{next}\ (x, \mathit{xs}, n+1) \\
& \qquad\qquad \mathit{else}\ (x,\ n,\ (x',\ \mathit{xs})))
\end{aligned}$$

The second step is to distribute the application of *continue* to the inner tail positions in the body of *next*. There are three such inner expressions in tail position—the first arm of the *case* expression and both arms of the *if* expression:

$$\begin{aligned}
= \quad & \{\mathit{distribute\ continue\ to\ inner\ tail\ positions}\} \\
& \lambda(x,\ \mathit{xs},\ n).\mathit{case}\ (x,\ \mathit{xs},\ n) \\
& \qquad \mathit{of}\ (x,\ \mathit{nil},\ n) \Rightarrow \mathit{continue}\ (x,\ n,\ ()) \\
& \qquad \mid (x,\ x' :: \mathit{xs},\ n) \Rightarrow\ \mathit{if}\ x = x' \\
& \qquad\qquad \mathit{then}\ \mathit{continue}\ (\mathit{next}\ (x, \mathit{xs}, n+1)) \\
& \qquad\qquad \mathit{else}\ \mathit{continue}\ (x,\ n,\ (x',\ \mathit{xs}))
\end{aligned}$$

The third step is to simplify by, e.g., inlining applications of *continue* to known arguments:

$$\begin{aligned}
= \quad & \{\mathit{inline\ applications\ of\ continue}\} \\
& \lambda(x,\ \mathit{xs},\ n).\mathit{case}\ (x,\ \mathit{xs},\ n) \\
& \qquad \mathit{of}\ (x,\ \mathit{nil},\ n) \Rightarrow (x,\ n) :: \mathit{nil} \\
& \qquad \mid (x,\ x' :: \mathit{xs},\ n) \Rightarrow\ \mathit{if}\ x = x' \\
& \qquad\qquad \mathit{then}\ \mathit{continue}\ (\mathit{next}\ (x, \mathit{xs}, n+1)) \\
& \qquad\qquad \mathit{else}\ (x,\ n) :: \mathit{continue}\ (\mathit{next}\ (x', \mathit{xs}, 1))
\end{aligned}$$

The fourth and final step is to use this abstraction to define a new recursive function next_c equal to $\mathit{continue} \circ \mathit{next}$, and to use it to replace remaining occurrences of $\mathit{continue} \circ \mathit{next}$. The auxiliary functions *next* and *continue* are no longer needed, and the fused run-length encoding function reads as follows:

$$\begin{aligned}
\mathit{next}_c : \alpha \times \mathit{List}(\alpha) \times \mathit{Nat} &\to \mathit{List}(\alpha \times \mathit{Nat}) \\
\mathit{next}_c\ (x, \mathit{nil}, n) &= (x,\ n) :: \mathit{nil} \\
\mathit{next}_c\ (x, x' :: \mathit{xs}, n) &= \ \mathit{if}\ x = x' \\
&\qquad \mathit{then}\ \mathit{next}_c\ (x, \mathit{xs}, n+1) \\
&\qquad \mathit{else}\ (x,\ n) :: \mathit{next}_c\ (x', \mathit{xs}, 1)
\end{aligned}$$

$$\begin{aligned}
\mathit{encode} : \mathit{List}(\alpha) &\to \mathit{List}(\alpha \times \mathit{Nat}) \\
\mathit{encode}\ \mathit{nil} &= \mathit{nil} \\
\mathit{encode}\ (x :: \mathit{xs}) &= \mathit{next}_c\ (x, \mathit{xs}, 1)
\end{aligned}$$

In an actual implementation, the first parameter of *next* and of $next_c$ would be lambda-dropped [23].

B Fusion of refocus and the context-based CPS transformation

We now calculate the definition of $\mathcal{R}_{\mathcal{T}_2}$, which is the fusion of the refocus function $\mathcal{R}$ and the context-based CPS transformation $\mathcal{T}_1$ from Section 2.1. We work with a version of $\mathcal{R}$ where the auxiliary function $\mathcal{R}'$ is inlined:

$$\begin{aligned}
&\mathcal{R} : \mathit{Terms} \times \mathit{Contexts} \to \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes}\\
\mathcal{R}(v,\ [\,]) &= v\\
\mathcal{R}(v_0,\ C[[\,]\ t_1]) &= \mathcal{R}(t_1,\ C[v_0\ [\,]])\\
\mathcal{R}(v_1,\ C[v_0\ [\,]]) &= (C, v_0\ v_1)\\
\mathcal{R}(t_0\ t_1,\ C) &= \mathcal{R}(t_0,\ C[[\,]\ t_1])
\end{aligned}$$

We follow the same steps as in Appendix A (as specified for multiargument uncurried functions [49]), starting with the composition of $\mathcal{T}_1$ and $\mathcal{R}$:

$$\begin{aligned}
&\lambda(t, C, k).\mathcal{T}_1\,(\mathcal{R}(t,\ C),\ k)\\
= \quad &\{\mathit{inline}\ \mathcal{R}\}\\
&\lambda(t, C, k).\mathcal{T}_1(\mathit{case}\ (t, C)\\
&\qquad\quad \mathit{of}\ (v, [\,]) \Rightarrow v\\
&\qquad\quad\ \mid (v_0, C[[\,]\ t_1]) \Rightarrow \mathcal{R}(t_1,\ C[v_0\ [\,]])\\
&\qquad\quad\ \mid (v_1, C[v_0\ [\,]]) \Rightarrow (C, v_0\ v_1)\\
&\qquad\quad\ \mid (t_0\ t_1, C) \Rightarrow \mathcal{R}(t_0,\ C[[\,]\ t_1]), k)\\
= \quad &\{\mathit{distribute}\ \mathcal{T}_1\,(-,\ k)\ \mathit{to\ inner\ tail\ positions}\}\\
&\lambda(t, C, k).\mathit{case}\ (t, C)\\
&\qquad\quad \mathit{of}\ (v, [\,]) \Rightarrow \mathcal{T}_1\,(v,\ k)\\
&\qquad\quad\ \mid (v_0, C[[\,]\ t_1]) \Rightarrow \mathcal{T}_1\,(\mathcal{R}(t_1,\ C[v_0\ [\,]]),\ k)\\
&\qquad\quad\ \mid (v_1, C[v_0\ [\,]]) \Rightarrow \mathcal{T}_1\,((C, v_0\ v_1),\ k)\\
&\qquad\quad\ \mid (t_0\ t_1, C) \Rightarrow \mathcal{T}_1\,(\mathcal{R}(t_0,\ C[[\,]\ t_1]),\ k)\\
= \quad &\{\mathit{inline\ two\ applications\ of}\ \mathcal{T}_1\}\\
&\lambda(t, C, k).\mathit{case}\ (t, C)\\
&\qquad\quad \mathit{of}\ (v, [\,]) \Rightarrow k\,(\mathcal{V}_1\,v)\\
&\qquad\quad\ \mid (v_0, C[[\,]\ t_1]) \Rightarrow \mathcal{T}_1\,(\mathcal{R}(t_1,\ C[v_0\ [\,]]),\ k)\\
&\qquad\quad\ \mid (v_1, C[v_0\ [\,]]) \Rightarrow (\mathcal{V}_1\,v_0)\,(\mathcal{V}_1\,v_1)\,(\mathcal{C}_1\,(C,\ k))\\
&\qquad\quad\ \mid (t_0\ t_1, C) \Rightarrow \mathcal{T}_1\,(\mathcal{R}(t_0,\ C[[\,]\ t_1]),\ k)
\end{aligned}$$

We then create a new recursive function $\mathcal{R}_{\mathcal{T}_2}$ to use in place of the composition of $\mathcal{T}_1$ and $\mathcal{R}$ (we rename $\mathcal{V}_1$ to $\mathcal{V}_2$ and $\mathcal{C}_1$ to $\mathcal{C}_2$, just as in Section 2.1):

$$\begin{aligned}
&\mathcal{R}_{\mathcal{T}_2} : \mathit{Terms} \times \mathit{Contexts} \times \mathit{Variables} \to \mathit{Terms}\\
\mathcal{R}_{\mathcal{T}_2}(v,\ [\,],\ k) &= k\,(\mathcal{V}_2\,v)\\
\mathcal{R}_{\mathcal{T}_2}(v_0,\ C[[\,]\ t_1],\ k) &= \mathcal{R}_{\mathcal{T}_2}(t_1,\ C[v_0\ [\,]],\ k)\\
\mathcal{R}_{\mathcal{T}_2}(v_1,\ C[v_0\ [\,]],\ k) &= (\mathcal{V}_2\,v_0)\,(\mathcal{V}_2\,v_1)\,(\mathcal{C}_2\,(C,\ k))\\
\mathcal{R}_{\mathcal{T}_2}(t_0\ t_1,\ C,\ k) &= \mathcal{R}_{\mathcal{T}_2}(t_0,\ C[[\,]\ t_1],\ k)
\end{aligned}$$

Inlining the auxiliary function $\mathcal{R}'_{\mathcal{T}_2}$ in the definition of $\mathcal{R}_{\mathcal{T}_2}$ from Section 2.1 yields this definition.

References

[1] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1986.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[3] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[4] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[5] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. CPS translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, 1999.

[6] Nick Benton and Andrew Kennedy. Monads, effects, and transformations. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999.

[7] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the technical report BRICS RS-06-3.

[8] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Technical Report BRICS RS-06-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2006. To appear in Theoretical Computer Science (extended version).

[9] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Technical Report BRICS RS-06-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006. Accepted for publication at TOPLAS.

[10] Roel Bloo, Fairouz Kamareddine, and Rob Nederpelt. The Barendregt cube with definitions and generalised reduction. *Information and Computation*, 126(2):123–143, 1996.

[11] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. *Journal of Functional Programming*, 13(5):867–904, 2003. A preliminary version was presented at the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000).

[12] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[13] Olivier Danvy, editor. *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, University of Aarhus, Paris, France, January 1997.

[14] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, volume 124(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[15] Olivier Danvy. Refunctionalization at work. In *Preliminary proceedings of the 8th International Conference on Mathematics of Program Construction (MPC '06)*, Kuressaare, Estonia, July 2006. Invited talk.

[16] 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.

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

[18] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[19] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[20] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science*, 308(1-3):239–257, 2003. A preliminary version was presented at the Fifth International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002).

[21] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[22] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. *Information Processing Letters*, 94(5):217–224, 2005. Extended version available as the research report BRICS RS-04-39.

[23] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[24] Olivier Danvy and Carolyn L. Talcott, editors. *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW'92)*, Technical report STAN-CS-92-1426, Stanford University, San Francisco, California, June 1992.

[25] Philippe de Groote. A CPS-translation of the λμ-calculus. In Sophie Tison, editor, *19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, number 787 in Lecture Notes in Computer Science, pages 47–58, Edinburgh, Scotland, April 1994. Springer-Verlag.

[26] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, August 1987.

[27] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. <`http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`>, 1989-2003.

[28] Andrzej Filinski. An extensional CPS transform (preliminary report). In Sabry [59], pages 41–46.

[29] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. Available at <`http://www.brics.dk/~hosc/vol06/03-fischer.html`>. A preliminary version was presented at the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[30] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[31] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13:21–51, 1991.

[32] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[33] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[34] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[35] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(3/4):361–380, 1993.

[36] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[37] 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.

[38] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

[39] Richard A. Kelsey. *Compilation by Program Transformation*. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, May 1989. Research Report 702.

[40] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, February 1988. Research Report 632.

[41] Jakov Kučan. Retraction approach to CPS transform. *Higher-Order and Symbolic Computation*, 11(2):145–175, 1998.

[42] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[43] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina, January 1993. ACM Press.

[44] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer-Verlag.

[45] Kevin Millikin. A new approach to one-pass transformations. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP 2005)*, pages 252–264, Tallinn, Estonia, September 2005. Institute of Cybernetics at Tallinn Technical University. Granted the best student-paper award of TFP 2005.

[46] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[47] Lasse R. Nielsen. A selective CPS transformation. In Stephen Brookes and Michael Mislove, editors, *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 201–222, Aarhus, Denmark, May 2001. Elsevier Science Publishers.

[48] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[49] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, New York, NY, USA, January 2007. ACM Press.

[50] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

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

[52] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[53] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, England, 1996.

[54] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2/3):161–180, 2002.

[55] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [57].

[56] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[57] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[58] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, August 1994. Technical report 94-242.

[59] Amr Sabry, editor. *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, Technical report 545, Computer Science Department, Indiana University, London, England, January 2001.

[60] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[61] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 29, No 6, pages 1–12, Orlando, Florida, June 1994. ACM Press.

[62] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997. A preliminary version was presented at the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP 1996).

[63] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[64] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 2007. Journal version of [63]. To appear.

[65] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[66] Guy L. Steele. Rabbit: A compiler for Scheme. M. Sc. thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[67] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [70].

[68] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1997. ECS-LFCS-97-376.

[69] Hayo Thielecke, editor. *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, Venice, Italy, January 2004.

[70] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.

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

[72] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

[73] Steve Zdancewic and Andrew Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2/3):209–234, 2002.

Recent BRICS Report Series Publications

RS-07-6 Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. *On One-Pass CPS Transformations*. March 2007. ii+19 pp. Theoretical Pearl to appear in the *Journal of Functional Programming*. Revised version of BRICS RS-02-3.

RS-07-5 Luca Aceto, Silvio Capobianco, and Anna Ingólfsdóttir. *On the Existence of a Finite Base for Complete Trace Equivalence over BPA with Interrupt*. February 2007. 26 pp.

RS-07-4 Kristian Støvring and Søren B. Lassen. *A Complete, Co-Inductive Syntactic Theory of Sequential Control and State*. February 2007. 36 pp. Appears in the proceedings of POPL 2007, p. 161–172.

RS-07-3 Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir. *Ready To Preorder: Get Your BCCSP Axiomatization for Free!* February 2007. 37 pp.

RS-07-2 Luca Aceto and Anna Ingólfsdóttir. *Characteristic Formulae: From Automata to Logic*. January 2007. 18 pp.

RS-07-1 Daniel Andersson. *HIROIMONO is NP-complete*. January 2007. 8 pp.

RS-06-19 Michael David Pedersen. *Logics for The Applied π Calculus*. December 2006. viii+111 pp.

RS-06-18 Małgorzata Biernacka and Olivier Danvy. *A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*. dec 2006. iii+39 pp. Extended version of an article to appear in TCS. Revised version of BRICS RS-05-22.

RS-06-17 Olivier Danvy and Kevin Millikin. *A Rational Deconstruction of Landin's J Operator*. December 2006. ii+37 pp. Revised version of BRICS RS-06-4. A preliminary version appears in the proceedings of IFL 2005, LNCS 4015:55–73.

RS-06-16 Anders Møller. *Static Analysis for Event-Based XML Processing*. October 2006. 16 pp.

RS-06-15 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. *A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*. October 2006. ii+28 pp. Revised version of BRICS RS-05-16.