JFP **17** (6): 793–812, 2007. © 2007 Cambridge University Press
doi:10.1017/S0956796807006387 Printed in the United Kingdom

On one-pass CPS transformations

OLIVIER DANVY, KEVIN MILLIKIN,
and LASSE R. NIELSEN

*BRICS, Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark*
(*e-mail:* `danvy,kmillikin,lrn@brics.dk`)

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 lambda-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.

1 Introduction

Transforming functional programs into continuation-passing style (CPS) is a classical topic, with a long publication history (Reynolds 1972; Strachey & Wadsworth 1974; Plotkin 1975; Steele 1978; Meyer & Wand 1985; Felleisen 1987; Kranz 1988; Appel & Jim 1989; Kelsey 1989; Griffin 1990; Fradet & Le Métayer 1991; Shivers 1991; Wand 1991; Danvy & Filinski 1992; Danvy & Lawall 1992; Danvy & Talcott 1992; Fischer 1993; Harper & Lillibridge 1993; Lawall & Danvy 1993; Reynolds 1993; Sabry & Felleisen 1993; Danvy 1994; de Groote 1994; Hatcliff 1994; Lawall 1994; Sabry 1994; Sabry & Felleisen 1994; Danvy 1997; Sabry & Wadler 1997; Thielecke 1997; Kučan 1998; Barthe *et al.* 1999; Filinski 2001; Nielsen 2001a; Sabry 2001; Reppy 2002; Zdancewic & Myers 2002; Damian & Danvy 2003; Danvy & Nielsen 2003; Thielecke 2004; Danvy & Nielsen 2005; Millikin 2005; Biernacki *et al.* 2006; Shan 2007)[1] including chapters in programming-languages textbooks (Appel 1992; Queinnec 1996; Friedman *et al.* 2001), 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.

[1] Among many others.

In this article, we bridge the two methodologically distinct CPS transformations described in the textbooks mentioned above. The first one, presented by Appel (1992) and by Queinnec (1996), is higher-order, and proceeds by recursive descent over the source program in a compositional way. The other one, presented by Friedman *et al.* (2001), 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 (Plotkin 1975; Steele 1978; Danvy & Filinski 1992; Sabry & Felleisen 1993). 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 (Plotkin 1981) introduced in Felleisen's Ph.D. thesis (Felleisen 1987) 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:

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

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 $\mathscr{D}$ that maps a value term to itself and a non-value term to a decomposition. There are many ways to define the $\mathscr{D}$ function, which is usually not shown in the literature. We use the following one here:

$$\begin{array}{rcl}
\mathscr{D} : \mathit{Terms} & \to & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathscr{D}\,t & = & \mathscr{D}'(t, [\,]) \\[1em]
\mathscr{D}' : \mathit{Terms} \times \mathit{Contexts} & \to & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathscr{D}'(v, C) & = & \mathscr{D}'_{\mathrm{aux}}(C, v) \\
\mathscr{D}'(t_0\,t_1, C) & = & \mathscr{D}'(t_0, C[[\,]\,t_1]) \\[1em]
\mathscr{D}'_{\mathrm{aux}} : \mathit{Contexts} \times \mathit{Values} & \to & \mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes} \\
\mathscr{D}'_{\mathrm{aux}}([\,], v) & = & v \\
\mathscr{D}'_{\mathrm{aux}}(C[[\,]\,t_1], v_0) & = & \mathscr{D}'(t_1, C[v_0\,[\,]]) \\
\mathscr{D}'_{\mathrm{aux}}(C[v_0\,[\,]], v_1) & = & (C, v_0\,v_1)
\end{array}$$

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

$$\begin{aligned}
\mathscr{D}'(v, [\,]) &= v \\
\mathscr{D}'(v_0, C[[\,]\, t_1]) &= \mathscr{D}'(t_1, C[v_0\, [\,]]) \\
\mathscr{D}'(v_1, C[v_0\, [\,]]) &= (C, v_0\, v_1) \\
\mathscr{D}'(t_0\, t_1, C) &= \mathscr{D}'(t_0, C[[\,]\, t_1])
\end{aligned}$$

but we prefer to keep it since as stated above, this definition is in defunctionalized form (Danvy 2004): the contexts form the data type of a defunctionalized function and $\mathscr{D}'_{\text{aux}}$ forms its apply function (Reynolds 1972; Danvy & Nielsen 2001). We will exploit this property in Section 2.2.

Conversely, one can also define a total function $\mathscr{P}$ 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 $\mathscr{P}$ function is straighforwardly defined by structural induction over its first argument:

$$\begin{aligned}
\mathscr{P} : \mathit{Contexts} \times \mathit{Terms} &\rightarrow \mathit{Terms} \\
\mathscr{P}([\,], t) &= t \\
\mathscr{P}(C[v_0\, [\,]], t_1) &= \mathscr{P}(C, v_0\, t_1) \\
\mathscr{P}(C[[\,]\, t_1], t_0) &= \mathscr{P}(C, t_0\, t_1)
\end{aligned}$$

In essence, and as envisioned by Sabry and Felleisen (1993), 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 (1993) for reasoning about programs in continuation-passing style.

Prerequisites: We assume a basic familiarity with the λ-calculus (Barendregt 1984), with reduction semantics (Felleisen 1987; Felleisen & Flatt 1989–2003; Xiao *et al.* 2001; Danvy & Nielsen 2004), and with the notion of one-pass CPS transformation (Danvy & Filinski 1992; Sabry & Felleisen 1993). We also make use of Reynolds's defunctionalization, i.e., the data-structure representation of higher-order functions (Reynolds 1972; Danvy & Nielsen 2001) and of its left inverse, refunctionalization (Danvy & Millikin, 2007).

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, and 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*)

$$\begin{array}{rcl}
\mathscr{T} : \mathit{Terms} \times \mathit{Variables} & \rightarrow & \mathit{Terms} \\
\mathscr{T}\,(v, k) & = & k\,(\mathscr{V}\,v) \\
\mathscr{T}\,(C[v_0\,v_1], k) & = & (\mathscr{V}\,v_0)\,(\mathscr{V}\,v_1)\,(\mathscr{C}\,(C, k)) \\
\\
\mathscr{V} : \mathit{Values} & \rightarrow & \mathit{Values} \\
\mathscr{V}\,x & = & x \\
\mathscr{V}\,\lambda x \cdot t & = & \lambda x \cdot \lambda k \cdot \mathscr{T}\,(t, k) \\
 & & \text{where } k \text{ is fresh} \\
\\
\mathscr{C} : \mathit{Contexts} \times \mathit{Variables} & \rightarrow & \mathit{Values} \\
\mathscr{C}\,(C, k) & = & \lambda w \cdot \mathscr{T}\,(C[w], k) \\
 & & \text{where } w \text{ is fresh}
\end{array}$$

The CPS transformation of a program t is $\lambda k \cdot \mathscr{T}\,(t, k)$, 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 $\mathscr{T}$) and the *plugging* of an expression into a context (on the right-hand side of the definition of $\mathscr{C}$). Here is an explicit version of this definition, using $\mathscr{D}$ and $\mathscr{P}$ as defined in Section 1:

Definition 2 (*Explicit context-based CPS transformation*)

$$\begin{array}{rcl}
\mathscr{T} : (\mathit{Values} + \mathit{Contexts} \times \mathit{PotRedexes}) \times \mathit{Variables} & \rightarrow & \mathit{Terms} \\
\mathscr{T}\,(v, k) & = & k\,(\mathscr{V}\,v) \\
\mathscr{T}\,((C, v_0\,v_1), k) & = & (\mathscr{V}\,v_0)\,(\mathscr{V}\,v_1)\,(\mathscr{C}\,(C, k)) \\
\\
\mathscr{V} : \mathit{Values} & \rightarrow & \mathit{Values} \\
\mathscr{V}\,x & = & x \\
\mathscr{V}\,\lambda x \cdot t & = & \lambda x \cdot \lambda k \cdot \mathscr{T}\,(\mathscr{D}\,t, k) \\
 & & \text{where } k \text{ is fresh} \\
\\
\mathscr{C} : \mathit{Contexts} \times \mathit{Variables} & \rightarrow & \mathit{Values} \\
\mathscr{C}\,(C, k) & = & \lambda w \cdot \mathscr{T}\,(\mathscr{D}\,(\mathscr{P}(C, w)), k) \\
 & & \text{where } w \text{ is fresh}
\end{array}$$

The CPS transformation of a program t is $\lambda k \cdot \mathscr{T}\,(\mathscr{D}\,t, k)$, 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 (Danvy & Nielsen 2004), which is an overkill since $\mathscr{D}$ is always applied to the result of $\mathscr{P}$. (We write 'always' since $\mathscr{D}\, t = \mathscr{D}\,(\mathscr{P}([\,], t))$.)

Danvy and Nielsen (2004) and Nielsen (2001b) have shown that the composition of plugging and decomposition can be fused into a 'refocus' function $\mathscr{R}$ 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 2004; Danvy & Nielsen 2004*)
For any term t and context C, $\mathscr{D}\,(\mathscr{P}(C, t)) = \mathscr{D}'(t, C)$.

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

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

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

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

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

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

Definition 3 (*Context-based CPS transformation, refocused*)

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

$$\begin{array}{rcl} \mathscr{V}_1 : \mathit{Values} & \rightarrow & \mathit{Values} \\ \mathscr{V}_1\, x & = & x \\ \mathscr{V}_1\, \lambda x \cdot t & = & \lambda x \cdot \lambda k \cdot \mathscr{T}_1\,(\mathscr{R}(t, [\,]), k) \\ & & \text{where } k \text{ is fresh} \end{array}$$

$$\begin{array}{rcl} \mathscr{C}_1 : \mathit{Contexts} \times \mathit{Variables} & \rightarrow & \mathit{Values} \\ \mathscr{C}_1\,(C, k) & = & \lambda w \cdot \mathscr{T}_1\,(\mathscr{R}(w, C), k) \\ & & \text{where } w \text{ is fresh} \end{array}$$

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

In Definition 2, $\mathscr{D}$ was always applied to the result of $\mathscr{P}$. Similarly, in Definition 3, $\mathscr{T}_1$ is always applied to the result of $\mathscr{R}$. Ohori and Sasano (2007) have shown that the composition of functions such as $\mathscr{T}_1$ and $\mathscr{R}$ can be fused by 'fixed-point promotion' into a function $\mathscr{R}_{\mathscr{T}_2}$ in such a way that for any term t, context C, and continuation identifier k,

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

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

Definition 4 (*Context-based CPS transformation, fused*)

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

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

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

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

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

Because the contexts are solely consumed by the rules defining $\mathscr{R}'_{\mathscr{T}_2}$, 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 $\mathscr{R}'_{\mathscr{T}_2}$ 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 $\mathscr{R}_{\mathscr{T}_3}(\widehat{C},\ t,\ k)$ to equal $\mathscr{R}_{\mathscr{T}_2}(C,\ t,\ k)$, and we write $\mathscr{V}_3$ and $\mathscr{C}_3$ to denote the counterparts of $\mathscr{V}_2$ and $\mathscr{C}_2$ 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
$$\overline{\lambda} u \cdot \overline{\lambda} k \cdot k \mathbin{\underline{@}} (\mathscr{V}_3\, u),$$
corresponding to the first rule of $\mathscr{R}'_{\mathscr{T}_2}$;

- if C is refunctionalized as $\widehat{C}$ then $C[v_0\,[\,]]$ is refunctionalized as

  $$\overline{\lambda} u_1 \cdot \overline{\lambda} k \cdot (\mathscr{V}_3\, v_0)\, \underline{@}\, (\mathscr{V}_3\, u_1)\, \underline{@}\, (\mathscr{C}_3(\widehat{C}, k)),$$

  corresponding to the third rule of $\mathscr{R}'_{\mathscr{T}_2}$; and
- if C is refunctionalized as $\widehat{C}$ then $C[[\,]\,t_1]$ is refunctionalized as

  $$\overline{\lambda} u_0 \cdot \overline{\lambda} k \cdot \mathscr{R}_{\mathscr{T}_3}(t_1, \overline{\lambda} u_1 \cdot \overline{\lambda} k \cdot (\mathscr{V}_3\, u_0)\, \underline{@}\, (\mathscr{V}_3\, u_1)\, \underline{@}\, (\mathscr{C}_3(\widehat{C}, k)), k),$$

  corresponding to the second rule of $\mathscr{R}'_{\mathscr{T}_2}$.

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

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

Definition 5 (*Context-based CPS transformation, refunctionalized*)

$$\begin{array}{rcl}
\mathscr{T}_3 : \mathit{Terms} \times (\mathit{Values} \to \mathit{Terms}) & \to & \mathit{Terms} \\
\mathscr{T}_3(v, \kappa) & = & \kappa\, \overline{@}\, v \\
\mathscr{T}_3(t_0\, t_1, \kappa) & = & \mathscr{T}_3(t_0, \overline{\lambda} u_0 \cdot \mathscr{T}_3(t_1, \overline{\lambda} u_1 \cdot (\mathscr{V}_3\, u_0) \\
 & & \underline{@}\, (\mathscr{V}_3\, u_1)\, \underline{@}\, (\mathscr{C}_3\, \kappa))) \\
 & & \\
\mathscr{V}_3 : \mathit{Values} & \to & \mathit{Values} \\
\mathscr{V}_3\, x & = & x \\
\mathscr{V}_3\, \lambda x \cdot t & = & \underline{\lambda} x \cdot \underline{\lambda} k \cdot \mathscr{T}_3(t, \overline{\lambda} u \cdot k\, \underline{@}\, (\mathscr{V}_3\, u)) \\
 & & \text{where } k \text{ is fresh} \\
 & & \\
\mathscr{C}_3 : (\mathit{Values} \to \mathit{Terms}) & \to & \mathit{Values} \\
\mathscr{C}_3\, \kappa & = & \underline{\lambda} w \cdot \kappa\, \overline{@}\, w \\
 & & \text{where } w \text{ is fresh}
\end{array}$$

The CPS transformation of a program t is $\underline{\lambda} k \cdot \mathscr{T}_3(t, \overline{\lambda} u \cdot k\, \underline{@}\, (\mathscr{V}_3\, u))$, 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 $\mathscr{V}_3$ 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 $\mathscr{T}_3$, $\mathscr{V}_3$, and $\mathscr{C}_3$ are then as follows:

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

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

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

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

Danvy and Filinski (1990, 1992), Wand (1991) and Appel (1992), each discovered the following higher-order one-pass CPS transformation:

Definition 6 (Higher-order CPS transformation)

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

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

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

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

Let us defunctionalize this higher-order transformation. The type $\mathit{CValues} \rightarrow \mathit{CTerms}$ 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)

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

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

$$\begin{aligned}
\mathcal{T}_5 &: DTerms \rightarrow Fun \rightarrow CTerms \\
\mathcal{T}_5(v, f) &= apply_5(f, \mathcal{V}_5\, v) \\
\mathcal{T}_5(t_0\, t_1, f) &= \mathcal{T}_5(t_0, F_1\,(f, t_1))
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{C}_5 &: Fun \rightarrow CValues \\
\mathcal{C}_5\, f &= \underline{\lambda} w \cdot apply_5(f, w) \\
&\ \text{where } w \text{ is fresh}
\end{aligned}$$

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

We recognize the result as a refocused context-based CPS transformation in which 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]

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

$$apply : Fun \times DValues \rightarrow CTerms$$

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

[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, as in Peyton Jones (1987, pp. 245–247).

[3] In another context (Danvy 2004; Danvy & Nielsen 2004; Biernacka & Danvy, in press, 2006a) fixed-point promotion makes it possible to transform a ‘pre-abstract machine’ into a ‘staged abstract machine’.

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 $\lambda x \cdot f\ (g\ x)$ into the following one:

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

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

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

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

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

3.1 Making a context-based CPS transformation tail-conscious

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

$$\begin{array}{rcll} \mathscr{C} : \mathit{Contexts} \times \mathit{Variables} & \rightarrow & \mathit{Values} & \\ \mathscr{C}\,([\,], k) & = & k & \\ \mathscr{C}\,(C, k) & = & \lambda w \cdot \mathscr{T}\,(C[w], k) & \text{if } C \neq [\,] \\ & & \text{where } w \text{ is fresh} & \end{array}$$

One can then take the same steps as in Section 2.1 to obtain a tail-conscious higher-order CPS transformation similar to that of Danvy and Filinski (1992).

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 $\mathscr{T}_4$ a sum, i.e., either the continuation identifier (in case of source tail call), or a function.

$$\begin{array}{rl} \mathscr{T}_4 : & \mathit{DTerms} \times (\mathit{Variables} + \mathit{CValues} \rightarrow \mathit{CTerms}) \rightarrow \mathit{CTerms} \\ \mathscr{C}_4 : & \mathit{Variables} + \mathit{CValues} \rightarrow \mathit{CTerms} \rightarrow \mathit{CValues} \end{array}$$

(Alternatively, the definition of $\mathscr{T}_4$ 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 $\lambda x \cdot \lambda k \cdot t$ or $\lambda k \cdot \lambda x \cdot t$? Since Plotkin (1975) and Steele (1978), tradition has it to do the former, but the latter makes curried continuation-passing functions continuation transformers (Gordon 1979; Allison 1986). Because this order was first promoted in

Fischer's (1993) work,[4] putting continuations first is said to be "à la Fischer" and is used, e.g., by Fradet and Le Métayer (1991), by Sabry and Felleisen (1993), and by Reppy (2002). 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 $\lambda y \cdot f\ x$ is transformed into $\lambda k \cdot k\ (\lambda y \cdot \lambda k \cdot f\ x\ k)$, where the inner continuation can be η-reduced.

Tail-conscious CPS à la Fischer: A term containing nested applications such as $\lambda x \cdot f\ (g\ (h\ x))$ is transformed into $\lambda k \cdot k\ (\lambda k \cdot \lambda x \cdot h\ (\lambda w_1 \cdot g\ (\lambda w_2 \cdot f\ k\ w_2)\ w_1)\ x)$. 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:

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

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 are *value* η-reductions, i.e., that they do not alter the properties of CPS-transformed programs, namely simulation, indifference, and translation (Plotkin 1975; Hatcliff & Danvy 1997) 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 for both Plotkin and Fischer, uniformly.

5 CPS transformation with generalized reduction

5.1 Generalized reduction

In his Ph.D. thesis (Sabry & Felleisen 1993; Sabry 1994), Sabry considered β_{lift}, a generalized reduction that is most easily described using reduction contexts (Bloo

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

et al. 1996):

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

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:

$$((\overline{\lambda} k \cdot \underline{\lambda} x \cdot t'_0)\, \overline{@}\, c)\, \underline{@}\, v'_1 \longrightarrow_{\text{adm}} (\underline{\lambda} x \cdot t'_0[c/k])\, \underline{@}\, v'_1$$

(t'_0 is the CPS counterpart of t_0, v'_1 is the CPS counterpart of v_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:

$$((\underline{\lambda} x \cdot \overline{\lambda} k \cdot t'_0)\, \underline{@}\, v'_1)\, \overline{@}\, c \longrightarrow_{\text{adm}} (\underline{\lambda} x \cdot t'_0[c/k])\, \underline{@}\, v'_1$$

5.2 *Administrative generalized reduction*

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

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

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

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

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 (Danvy & Nielsen 2005). 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 (Danvy & Nielsen 2005). 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 (Appel 1992; Queinnec 1996; Friedman *et al.* 2001) 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* (Friedman *et al.* 2001) 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 (Moggi 1991; Flanagan *et al.* 1993; Hatcliff & Danvy 1994; Benton & Kennedy 1999).

Acknowledgements

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

Appendix

A Fixed-point promotion

We outline Ohori and Sasano's fixed-point promotion algorithm (Ohori & Sasano 2007) 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. In addition, if the rest of the list is nonempty, it also returns its head and tail:

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

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

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

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

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

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*:

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

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:

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

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

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

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:

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

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

In an actual implementation, the first parameter of *next* and of next_c would be lambda-dropped (Danvy & Schultz 2000).

B Fusion of refocus and the context-based CPS transformation

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

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

We follow the same steps as in Appendix A (as specified for multiargument uncurried functions (Ohori & Sasano 2007)), starting with the composition of $\mathscr{T}_1$ and $\mathscr{R}$:

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

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

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

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

References

Allison, L. (1986) *A Practical Introduction to Denotational Semantics.* New York, NY: Cambridge University Press.

Appel, A. W. (1992) *Compiling With Continuations.* New York, NY: Cambridge University Press.

Appel, A. W. & Jim, T. (1989) Continuation-passing, closure-passing style. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, O'Donnell, M. J. & Feldman, S. (eds), Austin, TX: ACM Press, pp. 293–302.

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics.* Rev. ed. Studies in Logic and the Foundation of Mathematics, vol. 103. Amsterdam, The Netherlands, North-Holland.

Barthe, G., Hatcliff, J. & Sørensen, M. H. (1999) CPS translations and applications: The cube and beyond. *Higher-Order Symbolic Comput.* **12**(2), 125–170.

Benton, N. & Kennedy, A. (1999, Sept.) Monads, effects, and transformations. In *Third International Workshop on Higher-Order Operational Techniques in Semantics.* Electronic Notes in Theoretical Computer Science, vol. 26, pp. 19–31. Elsevier, Paris, France.

Biernacka, M. & Danvy, O. (in press) A concrete framework for environment machines. *ACM Trans. Computat. Logic.* Available as the technical report BRICS RS-06-3.

Biernacka, M. & Danvy, O. (2006a, Dec.) *A Syntactic Correspondence Between Context-Sensitive Calculi and Abstract Machines. Theor. Comput. Sci.* **375**, 76–108. Extended version available as Research report BRICS RS-06-18, University of Aarhus, Dec. 2006.

Biernacki, D., Danvy, O. & Millikin, K. (2006b, Oct.) *A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations.* Technical Report BRICS RS-06-15. DAIMI, Aarhus, Denmark: Department of Computer Science, University of Aarhus. Accepted for publication at TOPLAS.

Bloo, R., Kamareddine, F. & Nederpelt, R. (1996) The Barendregt cube with definitions and generalised reduction. *Inform. Comput.* **126**(2), 123–143.

Damian, D. & Danvy, O. (2003) Syntactic accidents in program analysis: On the impact of the CPS transformation. *J. Funct. Programming* **13**(5), 867–904. A preliminary version was presented at the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000).

Danvy, O. (1994) Back to direct style. *Sci. Comput. Programming* **22**(3), 183–195. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

Danvy, O. (ed). (1997, Jan.) *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97).* Technical report BRICS NS-96-13. Aarhus, Denmark: University of Aarhus.

Danvy, O. (2004) From reduction-based to reduction-free normalization. In *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming* *(wrs'04)*, Antoy, S. & Toyama, Y. (eds), Electronic Notes in Theoretical Computer Science **124**(2). Aachen, Germany: Elsevier Science, pp. 79–100. Invited talk.

Danvy, O. & Millikin, K. (2007) Refunctionalization at work. In Research Report BRICS RS-07-7, University of Aarhus.

Danvy, O. & Filinski, A. (1990) Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Wand, M. (ed), Nice, France: ACM Press, pp. 151–160.

Danvy, O. & Filinski, A. (1992) Representing control, a study of the CPS transformation. *Math. Struct. Comput. Sci.* **2**(4), 361–391.

Danvy, O. & Lawall, J. L. (1992) Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, Clinger, W. (ed), LISP Pointers, Vol. V, No. 1. San Francisco, CA: ACM Press, pp. 299–310.

Danvy, O. & Nielsen, L. R. (2001) Defunctionalization at work. In *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, Søndergaard, H. (ed.) Firenze, Italy: ACM Press, pp. 162–174. Extended version available as the technical report BRICS RS-01-23.

Danvy, O. & Nielsen, L. R. (2003) A first-order one-pass CPS transformation. *Theor. Comput. Sci.* **308**(1–3), 239–257. A preliminary version was presented at the Fifth International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002).

Danvy, O. & Nielsen, L. R. (2004, Nov.). *Refocusing in Reduction Semantics.* Research Report BRICS RS-04-26. DAIMI, Aarhus, Denmark: Department of Computer Science, University of Aarhus. 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 **59**(4).

Danvy, O. & Nielsen, L. R. (2005) CPS transformation of beta-redexes. *Inform. Process. Lett.* **94**(5), 217–224. Extended version available as the research report BRICS RS-04-39.

Danvy, O. & Schultz, U. P. (2000) Lambda-dropping: Transforming recursive equations into programs with block structure. *Theor. Comput. Sci.* **248**(1–2), 243–287.

Danvy, O. & Talcott, C. L. (eds). (1992, June) *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW'92).* Technical report STAN-CS-92-1426, Stanford University.

de Groote, P. (1994) A CPS-translation of the $\lambda\mu$-calculus. In *19th Colloquium on Trees in Algebra and Programming (CAAP'94)*, Tison, S. (ed), Lecture Notes in Computer Science, No. 787. Edinburgh, Scotland: Springer-Verlag.

Felleisen, M. (1987, Aug.) *The Calculi of λ-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. Thesis. Bloomington, IN: Computer Science Department, Indiana University.

Felleisen, M. & Flatt, M. (1989–2003) *Programming Languages and Lambda Calculi.* Unpublished lecture notes. Available at: `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html` 2006.

Filinski, A. (2001, Jan.) An extensional CPS transform (preliminary report). *In:* (Sabry 2001).

Fischer, M. J. (1993). Lambda-calculus schemata. *Lisp Symbolic Comput.* **6**(3/4), 259–288. 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.

Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, Wall, D. W. (ed), SIGPLAN Notices, Vol. 28, No 6. Albuquerque, NM: ACM Press, pp. 237–247.

Fradet, P. & Le Métayer, D. (1991) Compilation of functional languages by program transformation. *ACM Trans. Programming Lang. Syst.* **13**, 21–51.

Friedman, D. P., Wand, M. & Haynes, C. T. (2001) *Essentials of Programming Languages, 2nd ed.* Cambridge, MA: The MIT Press.

Gordon, M. J. C. (1979) *The Denotational Description of Programming Languages.* Springer-Verlag.

Griffin, T. G. (1990) A formulae-as-types notion of control. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, Hudak, P. (ed), San Francisco, CA: ACM Press, pp. 47–58.

Harper, B. & Lillibridge, M. (1993) Polymorphic type assignment and CPS conversion. *Lisp Symbolic Comput.* **6**(3/4), 361–380. Corrigendum (2006) in *Higher-Order Symbolic Comput.* **16**(4): 401.

Hatcliff, J. (1994, June) *The Structure of Continuation-Passing Styles.* Ph.D. Thesis, Manhattan, KS: Department of Computing and Information Sciences, Kansas State University.

Hatcliff, J. & Danvy, O. (1994) A generic account of continuation-passing styles. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Boehm, H.-J. (ed), Portland, OR: ACM Press, pp. 458–471.

Hatcliff, J. & Danvy, O. (1997) Thunks and the λ-calculus. *J. Funct. Programming* **7**(3), 303–319.

Kelsey, R. A. (1989, May) *Compilation by Program Transformation.* Ph.D. Thesis. New Haven, CT: Computer Science Department, Yale University, Research Report 702.

Kranz, D. A. (1988, Feb.) *Orbit: An Optimizing Compiler for Scheme.* Ph.D. Thesis. New Haven, CT: Computer Science Department, Yale University, Research Report 632.

Kučan, J. (1998) Retraction approach to CPS transform. *Higher-Order Symbolic Comput.* **11**(2), 145–175.

Lawall, J. L. (1994, July) *Continuation Introduction and Elimination in Higher-Order Programming Languages.* Ph.D. Thesis. Bloomington, IN: Computer Science Department, Indiana University.

Lawall, J. L. & Danvy, O. (1993) Separating stages in the continuation-passing style transformation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Graham, S. L. (ed), Charleston, SC: ACM Press, pp. 124–136.

Meyer, A. R. & Wand, M. (1985) Continuation semantics in typed lambda-calculi (summary). In *Logics of Programs – Proceedings*, Parikh, R. (ed), Lecture Notes in Computer Science, No. 193. Brooklyn, NY: Springer-Verlag, pp. 219–224.

Millikin, K. (2005) A new approach to one-pass transformations. In *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP 2005)*, van Eekelen, M. (ed), Tallinn, Estonia: Institute of Cybernetics at Tallinn Technical University, pp. 252–264. Granted the best student-paper award of TFP 2005.

Moggi, E. (1991) Notions of computation and monads. *Inform. Comput.* **93**, 55–92.

Nielsen, L. R. (2001a) A selective CPS transformation. In *Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics*, Brookes, S. & Mislove, M. (eds), Electronic Notes in Theoretical Computer Science, vol. 45. Aarhus, Denmark: Elsevier Science Publishers, pp. 201–222.

Nielsen, L. R. (2001b, July) *A Study of Defunctionalization and Continuation-Passing Style.* Ph.D. Thesis. Aarhus, Denmark: BRICS PhD School, University of Aarhus, BRICS DS-01-7.

Ohori, A. & Sasano, I. (2007) Lightweight fusion by fixed point promotion. *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, Felleisen, M. (ed), Nice, France: ACM Press, pp. 143–154.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International.

Plotkin, G. D. (1975) Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.* **1**, 125–159.

Plotkin, G. D. (1981, Sept.) *A Structural Approach to Operational Semantics*. Tech. Rept. FN-19. DAIMI, Aarhus, Denmark: Department of Computer Science, University of Aarhus.

Queinnec, C. (1996) *Lisp in Small Pieces*. Cambridge, England: Cambridge University Press.

Reppy, J. (2002) Optimizing nested loops using local CPS conversion. *Higher-Order Symbolic Comput.* **15**(2/3), 161–180.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pp. 717–740. Reprinted in *Higher-Order Symbolic Comput.* **11**(4), 363–397, 1998, with a foreword (Reynolds 1998).

Reynolds, J. C. (1993) The discoveries of continuations. *Lisp Symbolic Comput.* **6**(3/4), 233–247.

Reynolds, J. C. (1998) Definitional interpreters revisited. *Higher-Order and Symbolic Comput.* **11**(4), 355–361.

Sabry, A. (1994, Aug.) *The Formal Relationship Between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. Ph.D. Thesis. Houston, TX: Computer Science Department, Rice University, Technical report 94-242.

Sabry, A. (ed). (2001, Jan.) *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (cw'01)*. Technical report 545, Computer Science Department, Indiana University.

Sabry, A. & Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp Symbolic Comput.* **6**(3/4), 289–360.

Sabry, A. & Felleisen, M. (1994) Is continuation-passing useful for data flow analysis? In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, Sarkar, V. (ed.), SIGPLAN Notices, Vol. 29, No 6. Orlando, FL: ACM Press, pp. 1–12.

Sabry, A. & Wadler, P. (1997) A reflection on call-by-value. *ACM Trans. Programming Lang. Sys.* **19**(6), 916–941. A preliminary version was presented at the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP 1996).

Shan, C.-C. (2004, Sept.) Shift to control. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Shivers, O. & Waddell, O. (eds), Technical Report TR600, Computer Science Department, Indiana University.

Shan, C.-C. (2007) A static simulation of dynamic delimited control. *Higher-Order Symbolic Comput.* **20**(4), Journal version of (Shan 2004).

Shivers, O. 1991 (May) *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. Ph.D. Thesis. Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-91-145.

Steele, G. L. (1978, May) *Rabbit: A Compiler for Scheme*. M.Sc. thesis. Cambridge: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report AI-TR-474.

Strachey, C. & Wadsworth, C. P. (1974) *Continuations: A Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG-11. Oxford, England: Oxford University Computing Laboratory, Programming Research Group. Reprinted in *Higher-Order Symbolic Comput.* **13**(1/2), 135–152, 2000, with a foreword (Wadsworth 2000).

Thielecke, H. (1997) *Categorical Structure of Continuation Passing Style*. Ph.D. thesis. Edinburgh, Scotland: University of Edinburgh, ECS-LFCS-97-376.

Thielecke, H. (ed). (2004, Jan.) *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (cw'04)*. Technical report CSR-04-1, Department of Computer Science, Queen Mary's College.

Wadsworth, C. P. (2000) Continuations revisited. *Higher-Order Symbolic Comput* **13**(1/2), 131–133.

Wand, M. (1991) Correctness of procedure representations in higher-order assembly language. In *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, Brookes, S., Main, M., Melton, A., Mislove, M. & Schmidt, D. (eds), Lecture Notes in Computer Science, No. 598. Pittsburgh, PA: Springer-Verlag, pp. 294–311.

Xiao, Y., Sabry, A. & Ariola, Z. M. (2001) From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Comput*. **14**(4), 387–409.

Zdancewic, S. & Myers, A. (2002) Secure information flow via linear continuations. *Higher-Order Symbolic Comput*. **15**(2/3), 209–234.