Basic Research in Computer Science

On One-Pass CPS Transformations

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series **RS-02-3**

ISSN 0909-0878 **January 2002**

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
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
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/02/3/`

On One-Pass CPS Transformations

Olivier Danvy and Lasse R. Nielsen
BRICS *
Department of Computer Science
University of Aarhus †

January 17, 2001

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 due to Appel, Danvy and Filinski, and Wand, building on Plotkin's seminal work. The other is non-compositional and based on a syntactic theory of the λ-calculus, and is due to Sabry and Felleisen. To relate the two approaches, we use Church encoding, Reynolds's defunctionalization, and an implementation technique for syntactic theories, refocusing, developed in the second author's PhD thesis.

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

*Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {`danvy,lrn`}`@brics.dk`

Contents

1 Introduction

Transforming functional programs into continuation-passing style (CPS) is a classical topic, with a long publication history [2, 4, 8, 9, 11, 15, 17, 18, 20, 23, 24, 25, 26, 29, 30, 31, 32, 33, 35, 38, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 52],[1] including chapters in programming-languages textbooks [1, 21, 40], and many applications. Yet no standard algorithm for CPS transformation has emerged, and this lack contributes to maintaining continuations, CPS, and CPS transformations as mystifying artefacts 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 [1] and by Queinnec [40], is higher-order, and proceeds by recursive descent over the source program, compositionally. The other one, presented by Friedman, Wand, and Haynes [21], is context-based, and rewrites the source program incrementally, non-compositionally. Both transformations yield compact programs, i.e., without administrative redexes [11, 38, 46, 47]. 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 syntactic theories, a variant of Plotkin's structured operational semantics [39] introduced in Felleisen's PhD thesis [16] and based on the notion of evaluation contexts.

In a syntactic theory for the call-by-value λ-calculus, terms, values, and evaluation contexts are defined as follows.

$$
\begin{array}{rclcrcl}
e & \in & \mathit{Exp} & \qquad & e & ::= & v \mid e\, e \\
v & \in & \mathit{Val} & & v & ::= & x \mid \lambda x.e \\
x, k, w & \in & \mathit{Var} & & & & \\
E & \in & \mathit{EvCont} & & E & ::= & [\,] \mid E[v\,[\,]] \mid E[[\,]\,e]
\end{array}
$$

In essence, the context-based CPS transformation decomposes a source term into a context and an application of two values, CPS transforms the application, plugs a fresh variable in the context, and iterates. That is our starting point in Section 2.1. We then massage this 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 eta-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 eta-reductions enabled

[1] Among many others.

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 eta-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 [46].

Prerequisites: We assume a basic familiarity with the λ-calculus [3], with syntactic theories [14, 16, 51], and with the notion of one-pass CPS transformation [11, 46]. We also make use of Church encoding, i.e., the higher-order representation of data structures [7], and of Reynolds's defunctionalization, i.e., the data-structure representation of higher-order functions [13, 43].

2 Standard CPS transformation

2.1 From context-based to higher-order

The following CPS transformation repeatedly decomposes a source term into a context and the application of one value to another value, CPS transforms the application, and plugs a fresh variable in the context. This process continues until the source term is a value.

Definition 1 (Context-based CPS transformation)

$$
\begin{array}{rcl}
\mathcal{E} : \mathit{Exp} \times \mathit{Var} & \rightarrow & \mathit{Exp} \\
\mathcal{E}[\![v]\!]\,k & = & k\,\mathcal{V}[\![v]\!] \\
\mathcal{E}[\![E[v_0\,v_1]]\!]\,k & = & \mathcal{V}[\![v_0]\!]\,\mathcal{V}[\![v_1]\!]\,(\mathcal{C}[\![E]\!]\,k) \\
\\
\mathcal{V} : \mathit{Val} & \rightarrow & \mathit{Val} \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda k.\mathcal{E}[\![e]\!]\,k \\
 & & \textit{where } k \textit{ is fresh} \\
\\
\mathcal{C} : \mathit{EvCont} \times \mathit{Var} & \rightarrow & \mathit{Val} \\
\mathcal{C}[\![E]\!]\,k & = & \lambda w.\mathcal{E}[\![E[w]]\!]\,k \\
 & & \textit{where } w \textit{ is fresh}
\end{array}
$$

The CPS transformation of a complete program e is $\lambda k.\mathcal{E}[\![e]\!]\,k$, where k is fresh.
□

Implicit in Definition 1 are the *decomposition* of a non-value source expression into a context and an application of a value to another value (third line of the definition of $\mathcal{E}$, in the left-hand side) and the *plugging* of an expression in a context (second line of the definition of $\mathcal{C}$, in the right-hand side). If they are implemented literally, decomposition and plugging entail a time factor for each transformation step 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 quadratic in the size of the source program.

In another work [14, 36], we have shown that the composition of plugging and decomposition can be simplified into a *refocus* function that make the resulting CPS transformation operate in time linear in the size of the source program—or more precisely, in one pass. Intuitively, *refocus* maps an expression and a context into the next context and application of one value to another value, if there is any.

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

Definition 2 (Context-based CPS transformation, refocused)

$$
\begin{array}{rcl}
\mathit{refocus} : \mathit{Exp} \times \mathit{EvCont} & \rightarrow & \mathit{Val} + (\mathit{EvCont} \times \mathit{Exp}) \\
\mathit{refocus}[\![v,\, E]\!] & = & \mathit{refocus}'[\![E,\, v]\!] \\
\mathit{refocus}[\![e_0\, e_1,\, E]\!] & = & \mathit{refocus}[\![e_0,\, E[[\,]\, e_1]]\!] \\
\\
\mathit{refocus}' : \mathit{EvCont} \times \mathit{Val} & \rightarrow & \mathit{Val} + (\mathit{EvCont} \times \mathit{Exp}) \\
\mathit{refocus}'[\![[\,],\, v]\!] & = & [\![v]\!] \\
\mathit{refocus}'[\![E[[\,]\, e_1],\, v_0]\!] & = & \mathit{refocus}[\![e_1,\, E[v_0\, [\,]]]\!] \\
\mathit{refocus}'[\![E[v_0\, [\,]],\, v_1]\!] & = & [\![E,\ v_0\, v_1]\!] \\
\\
\mathcal{E} : (\mathit{Val} + (\mathit{EvCont} \times \mathit{Exp})) \times \mathit{Var} & \rightarrow & \mathit{Exp} \\
\mathcal{E}[\![v]\!]\, k & = & k\, \mathcal{V}[\![v]\!] \\
\mathcal{E}[\![E,\ v_0\, v_1]\!]\, k & = & \mathcal{V}[\![v_0]\!]\, \mathcal{V}[\![v_1]\!]\, (\mathcal{C}[\![E]\!]\, k) \\
\\
\mathcal{V} : \mathit{Val} & \rightarrow & \mathit{Val} \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda k.\mathcal{E}(\mathit{refocus}[\![e,\, [\,]]\!])\, k \\
 & & \textit{where } k \textit{ is fresh} \\
\\
\mathcal{C} : \mathit{EvCont} \times \mathit{Var} & \rightarrow & \mathit{Val} \\
\mathcal{C}[\![E]\!]\, k & = & \lambda w.\mathcal{E}(\mathit{refocus}[\![w,\, E]\!])\, k \\
 & & \textit{where } w \textit{ is fresh}
\end{array}
$$

The CPS transformation of a complete program e is $\lambda k.\mathcal{E}(\mathit{refocus}[\![e,\, [\,]]\!])\, k$*, where* k *is fresh.* □

Let us now fuse $\mathcal{E}$ and *refocus* into one function $\mathit{refocus}_{\mathcal{E}}$ in such a way that

$$\forall e, E, k \,.\, \mathcal{E}(\mathit{refocus}[\![e,\, E]\!])\, k = \mathit{refocus}_{\mathcal{E}}[\![e,\, E]\!]\, k.$$

A simple fold/unfold calculation yields the following CPS transformation.

Definition 3 (Context-based CPS transformation, fused)

$$
\begin{array}{rcl}
\mathit{refocus}_{\mathcal{E}} : (\mathit{Exp} \times \mathit{EvCont}) \times \mathit{Var} & \rightarrow & \mathit{Exp} \\
\mathit{refocus}_{\mathcal{E}}[\![v,\, E]\!]\, k & = & \mathit{refocus}'_{\mathcal{E}}[\![E,\, v]\!]\, k \\
\mathit{refocus}_{\mathcal{E}}[\![e_0\, e_1,\, E]\!]\, k & = & \mathit{refocus}_{\mathcal{E}}[\![e_0,\, E[[\,]\, e_1]]\!]\, k \\[1em]
\mathit{refocus}'_{\mathcal{E}} : (\mathit{EvCont} \times \mathit{Val}) \times \mathit{Var} & \rightarrow & \mathit{Exp} \\
\mathit{refocus}'_{\mathcal{E}}[\![[\,],\, v]\!]\, k & = & k\, \mathcal{V}[\![v]\!] \\
\mathit{refocus}'_{\mathcal{E}}[\![E[[\,]\, e_1],\, v_0]\!]\, k & = & \mathit{refocus}_{\mathcal{E}}[\![e_1,\, E[v_0\, [\,]]]\!]\, k \\
\mathit{refocus}'_{\mathcal{E}}[\![E[v_0\, [\,]],\, v_1]\!]\, k & = & \mathcal{V}[\![v_0]\!]\, \mathcal{V}[\![v_1]\!]\, (\mathcal{C}[\![E]\!]\, k) \\[1em]
\mathcal{V} : \mathit{Val} & \rightarrow & \mathit{Val} \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \lambda x.\lambda k.\mathit{refocus}_{\mathcal{E}}[\![e,\, [\,]]\!]\, k \\
& & \mathit{where}\ k\ \mathit{is\ fresh} \\[1em]
\mathcal{C} : \mathit{EvCont} \times \mathit{Var} & \rightarrow & \mathit{Val} \\
\mathcal{C}[\![E]\!]\, k & = & \lambda w.\mathit{refocus}_{\mathcal{E}}[\![w,\, E]\!]\, k \\
& & \mathit{where}\ w\ \mathit{is\ fresh}
\end{array}
$$

The CPS transformation of a complete program e is $\lambda k.\mathit{refocus}_{\mathcal{E}}[\![[\,],\, e]\!]\, k$, where k is fresh. □

As the last step of the derivation, let us Church-encode the contexts, which are constructed in the calls to $\mathit{refocus}_{\mathcal{E}}$ and consumed in each of the rules defining $\mathit{refocus}'_{\mathcal{E}}$.

Under the assumption that E is Church-encoded as $\widetilde{E}$, and for any e and k, we define $\mathit{refocus}_{\widetilde{\mathcal{E}}}[\![\widetilde{E},\, e]\!]\, k$ to equal $\mathit{refocus}_{\mathcal{E}}[\![E,\, e]\!]\, k$. We write $\widetilde{\mathcal{V}}$ and $\widetilde{\mathcal{C}}$ to denote the counterparts of $\mathcal{V}$ and $\mathcal{C}$ on Church-encoded contexts, and we overline λ and the infix operator @ for the static abstractions and applications corresponding to Church encoding; 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 Church-encoded as
$$\overline{\lambda} k.\overline{\lambda} u.k \mathbin{\underline{@}} \widetilde{\mathcal{V}}[\![u]\!],$$
corresponding to the first rule of $\mathit{refocus}'_{\mathcal{E}}$ in Definition 3;

- if E is Church-encoded as $\widetilde{E}$ then $E[v_0\, [\,]]$ is Church-encoded as
$$\overline{\lambda} k.\overline{\lambda} u_1.\widetilde{\mathcal{V}}[\![v_0]\!] \mathbin{\underline{@}} \widetilde{\mathcal{V}}[\![u_1]\!] \mathbin{\underline{@}} (\widetilde{\mathcal{C}}[\![\widetilde{E}]\!]\, k),$$
corresponding to the third rule of $\mathit{refocus}'_{\mathcal{E}}$; and

- if E is Church-encoded as $\widetilde{E}$ then $E[[\,]\,e_1]$ is Church-encoded as

$$\overline{\lambda} k.\overline{\lambda} u_0.\mathit{refocus}_{\widetilde{\mathcal{E}}}[\![e_1,\ \overline{\lambda} k.\overline{\lambda} u_1.\widetilde{\mathcal{V}}[\![u_0]\!]\ \underline{@}\ \widetilde{\mathcal{V}}[\![u_1]\!]\ \underline{@}\ (\widetilde{\mathcal{C}}[\![\widetilde{E}]\!]\ k)]\!]\ k.$$

corresponding to the second rule of $\mathit{refocus}'_{\mathcal{E}}$.

The interpretation of contexts performed by $\mathit{refocus}'_{\mathcal{E}}$ is now part of the Church encoding. There is thus no need for the definition of $\mathit{refocus}'_{\mathcal{E}}$ and we omit it.

In the definition below, instead of $\mathit{refocus}_{\widetilde{\mathcal{E}}}$ that operates on e, $\widetilde{E}$, and k, we define a function $\mathcal{E}$ operating on e and on $\widetilde{E}\,\overline{@}\,k$, so that $\mathit{refocus}_{\widetilde{\mathcal{E}}}[\![e,\ \widetilde{E}]\!]\ k = \mathcal{E}[\![e]\!]\ (\widetilde{E}\,\overline{@}\,k)$. The result is a higher-order CPS transformation.

Definition 4 (Context-based CPS transformation, Church-encoded)

$$\begin{array}{rcl}
\mathcal{E} : \mathit{Exp} \times (\mathit{Val} \to \mathit{Exp}) & \to & \mathit{Exp} \\
\mathcal{E}[\![v]\!]\ \kappa & = & \kappa\,\overline{@}\,v \\
\mathcal{E}[\![e_0\ e_1]\!]\ \kappa & = & \mathcal{E}[\![e_0]\!]\ \overline{\lambda} u_0.\mathcal{E}[\![e_1]\!]\ \overline{\lambda} u_1.\mathcal{V}[\![u_0]\!]\ \underline{@}\ \mathcal{V}[\![u_1]\!]\ \underline{@}\,\mathcal{C}(\kappa) \\
\\
\mathcal{V} : \mathit{Val} & \to & \mathit{Val} \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{E}[\![e]\!]\ \overline{\lambda} u.k\ \underline{@}\ \mathcal{V}[\![u]\!] \\
 & & \textit{where } k \textit{ is fresh} \\
\\
\mathcal{C} : (\mathit{Val} \to \mathit{Exp}) & \to & \mathit{Val} \\
\mathcal{C}(\kappa) & = & \underline{\lambda} w.\kappa\,\overline{@}\,w \\
 & & \textit{where } w \textit{ is fresh}
\end{array}$$

The CPS transformation of a complete program e is $\underline{\lambda} k.\mathcal{E}[\![e]\!]\ \overline{\lambda} u.k\ \underline{@}\ \mathcal{V}[\![u]\!]$, 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 Church-encodings that λ-abstract the contents of the double brackets. 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 *DExp* and *DVal* for the syntactic domains of source, direct-style expressions and values and *CExp* and *CVal* for the syntactic domains of target, CPS expressions and values. The types of $\mathcal{E}$, $\mathcal{V}$, and $\mathcal{C}$ are then as follows:

$$\begin{array}{rcl}
\mathcal{E} : \mathit{DExp} \times (\mathit{DVal} \to \mathit{CExp}) & \to & \mathit{CExp} \\
\mathcal{V} : \mathit{DVal} & \to & \mathit{CVal} \\
\mathcal{C} : (\mathit{DVal} \to \mathit{CExp}) & \to & \mathit{CVal}
\end{array}$$

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

$$\begin{array}{rcl} \mathcal{E} : \mathit{DExp} \times (\mathit{CVal} \to \mathit{CExp}) & \to & \mathit{CExp} \\ \mathcal{C} : (\mathit{CVal} \to \mathit{CExp}) & \to & \mathit{CVal} \end{array}$$

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 [1], Danvy and Filinski [10, 11], and Wand [50] each discovered the following higher-order one-pass CPS transformation.

Definition 5 (Higher-order CPS transformation)

$$\begin{array}{rcl} \mathcal{E} : \mathit{DExp} \times (\mathit{CVal} \to \mathit{CExp}) & \to & \mathit{CExp} \\ \mathcal{E}[\![v]\!]\,\kappa & = & \kappa \,\overline{@}\, \mathcal{V}[\![v]\!] \\ \mathcal{E}[\![e_0\; e_1]\!]\,\kappa & = & \mathcal{E}[\![e_0]\!]\,\overline{\lambda} u_0.\mathcal{E}[\![e_1]\!]\,\overline{\lambda} u_1.u_0 \,\underline{@}\, u_1 \,\underline{@}\, \mathcal{C}(\kappa) \\ \\ \mathcal{V} : \mathit{DVal} & \to & \mathit{CVal} \\ \mathcal{V}[\![x]\!] & = & x \\ \mathcal{V}[\![\lambda x.e]\!] & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{E}[\![e]\!]\,\overline{\lambda} u.k \,\underline{@}\, u \\ & & \textit{where } k \textit{ is fresh} \\ \\ \mathcal{C} : (\mathit{CVal} \to \mathit{CExp}) & \to & \mathit{CVal} \\ \mathcal{C}(\kappa) & = & \underline{\lambda} w.\kappa \,\overline{@}\, w \\ & & \textit{where } w \textit{ is fresh} \end{array}$$

The CPS transformation of a complete program e is $\underline{\lambda} k.\mathcal{E}[\![e]\!]\,\overline{\lambda} u.k \,\underline{@}\, u$, where k is fresh. □

Let us defunctionalize this higher-order transformation [13, 43]. The type $\mathit{CVal} \to \mathit{CExp}$ is inhabited by instances of three λ-abstractions (the overlined λ-abstractions in Definition 5). It therefore gives rise to a data type with three constructors (written below as in ML) and its associated apply function.

The corresponding defunctionalized CPS transformation reads as follows.

Definition 6 (Higher-order CPS transformation, defunctionalized)

$$\begin{array}{rcl} \textit{datatype Fun} & = & F_0 \textit{ of } \mathit{Var} \\ & | & F_1 \textit{ of } \mathit{Fun} \times \mathit{DExp} \\ & | & F_2 \textit{ of } \mathit{Fun} \times \mathit{CVal} \end{array}$$

$$
\begin{array}{rcl}
apply : Fun \times CVal & \rightarrow & CExp \\
apply(F_0(k),\, u) & = & k \mathbin{\underline{@}} u \\
apply(F_1(f,\, e_1),\, u_0) & = & \mathcal{E}[\![e_1]\!]\,(F_2(f,\, u_0)) \\
apply(F_2(f,\, u_0),\, u_1) & = & u_0 \mathbin{\underline{@}} u_1 \mathbin{\underline{@}} \mathcal{C}(f) \\
\\
\mathcal{E} : DExp \times Fun & \rightarrow & CExp \\
\mathcal{E}[\![v]\!]\, f & = & apply(f,\, \mathcal{V}[\![v]\!]) \\
\mathcal{E}[\![e_0\, e_1]\!]\, f & = & \mathcal{E}[\![e_0]\!]\,(F_1(f,\, e_1)) \\
\\
\mathcal{V} : DVal & \rightarrow & CVal \\
\mathcal{V}[\![x]\!] & = & x \\
\mathcal{V}[\![\lambda x.e]\!] & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{E}[\![e]\!]\,(F_0(k)) \\
 & & \textit{where } k \textit{ is fresh} \\
\\
\mathcal{C} : Fun & \rightarrow & CVal \\
\mathcal{C}(f) & = & \underline{\lambda} w.apply(f,\, w) \\
 & & \textit{where } w \textit{ is fresh}
\end{array}
$$

The CPS transformation of a complete program e is $\underline{\lambda} k.\mathcal{E}[\![e]\!]\,(F_0(k))$, where k is fresh. □

We recognize the result as a refocused context-based CPS transformation where the contexts hold elements of $CVal$ instead of elements of $DVal$. The data type Fun plays the role of the evaluation contexts (indexing each empty context with a continuation identifier), $apply$ plays the role of $refocus'_{\widetilde{\mathcal{E}}}$, and $\mathcal{E}$ plays the role of $refocus_{\widetilde{\mathcal{E}}}$.

Alternatively, we can defunctionalize the CPS transformation of Definition 5 so that the data type and the type of its apply function read as follows.[2]

$$
\begin{array}{rcl}
\textit{datatype Fun} & = & F_0 \textit{ of } Var \\
 & | & F_1 \textit{ of } Fun \times DExp \\
 & | & F_2 \textit{ of } Fun \times DVal \\
\\
apply : Fun \times DVal & \rightarrow & CExp
\end{array}
$$

We then obtain the CPS transformation of Definition 3.

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

[2] This latitude in defining a data type is similar to the latitude of choosing maximally vs. minimally free expressions in super-combinator conversion [37, Section 15.2].

same goal, the two approaches have been developed independently and have always been reported separately in the literature.

The tools we have used to bridge the two CPS transformations are refocusing, unfolding and folding, Church encoding, and defunctionalization. Refocusing is the key tool to make the context-based CPS transformation operate in one pass. Unfolding and folding are a basic method for semantics-based program manipulation. Church encoding and defunctionalization are essentially inverse changes of representation between the first-order world and the higher-order world.

3 Tail-conscious CPS transformation

The CPS transformations of Section 2 generate one eta-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 eta-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 1 can be refined as follows to make it tail-conscious:

$$\begin{array}{rcll} \mathcal{C} : \mathit{EvCont} \times \mathit{Var} & \rightarrow & \mathit{Val} & \\ \mathcal{C}[\![[\,]]\!]\,k & = & k & \\ \mathcal{C}[\![E]\!]\,k & = & \lambda w.\mathcal{E}[\![E[w]]\!]\,k & \text{if } E \neq [\,] \\ & & \text{where } w \text{ is fresh} & \end{array}$$

One can then take the same steps as in Section 2.1 to obtain a tail-conscious higher-order CPS transformation similar to Danvy and Filinski's [11].

3.2 Making a higher-order CPS transformation tail-conscious

The specification in Definition 5 can be refined to make it tail-conscious. The idea is to make the second parameter of $\mathcal{E}$ a sum, i.e., either the continuation identifier (in case of source tail call), or a function.

$$\begin{array}{rcl} \mathcal{E} : \mathit{DExp} \times (\mathit{Var} + (\mathit{CVal} \rightarrow \mathit{CExp})) & \rightarrow & \mathit{CExp} \\ \mathcal{C} : \mathit{Var} + (\mathit{CVal} \rightarrow \mathit{CExp}) & \rightarrow & \mathit{CVal} \end{array}$$

(Alternatively, the definition of $\mathcal{E}$ 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 $\lambda x.\lambda k.e$ or $\lambda k.\lambda x.e$? Since Plotkin [38] and Steele [47], tradition has it to do the former, but the latter makes curried continuation-passing functions continuation transformers [22]. Because this order was first promoted in Fischer's work [18],[3] putting continuations first is said to be "à la Fischer" and is used, e.g., by Fradet and Le Métayer [20], by Reppy [41], and by Sabry and Felleisen [46]. 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 eta-reductions, which are a source of compactness in CPS programs.

Tail-conscious CPS à la Plotkin: In a λ-abstraction, a tail call where subterms 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 eta-reduced.

Tail-conscious CPS à la Fischer: A term with nested applications such as $\lambda x.f\ (g\ (h\ x))$ is transformed as follows:

$$\lambda k.k\ (\lambda k.\lambda x.h\ (\lambda w_1.g\ (\lambda w_2.f\ k\ w_2)\ w_1)\ x)$$

In this CPS term, the parameter of each continuation can be administratively eta-reduced, producing the following term:

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

(Indeed even x can be eta-reduced.)

As the two examples illustrate, a curried CPS à la Plotkin makes it possible to eta-reduce continuation identifiers for some source λ-abstractions, whereas a curried CPS à la Fischer makes it possible to eta-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 eta-reductions.

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

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

that administrative eta-reductions are *value* eta-reductions, i.e., that they do not alter the properties of CPS-transformed programs, namely simulation, indifference, and translation [28, 38] as well as termination.

At any rate, the current agreement in the continuation community is that administrative eta-reductions bring more trouble than benefits. In fact, for uncurried CPS, neither flavor provides any extra opportunity for administrative eta-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 [44, 46], Sabry considered β_{lift}, a generalized reduction that is most easily described using evaluation contexts [6]:

$$E[(\lambda x.e_0)\; e_1] \quad \longrightarrow_{\beta_{\mathit{lift}}} \quad (\lambda x.E[e_0])\; e_1$$

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:

$$((\overline{\lambda} k.\underline{\lambda} x.e'_0)\,\overline{@}\,c)\,\underline{@}\,v'_1 \quad \longrightarrow_{adm} \quad (\underline{\lambda} x.e'_0[c/k])\,\underline{@}\,v'_1$$

(e'_0 is the CPS counterpart of e_0, v'_1 is the CPS counterpart of e_1, and c represents E.)

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:

$$((\underline{\lambda} x.\overline{\lambda} k.e'_0)\,\underline{@}\,v'_1)\,\overline{@}\,c \quad \longrightarrow_{adm} \quad (\underline{\lambda} x.e'_0[c/k])\,\underline{@}\,v'_1$$

5.2 Administrative generalized reduction

Integrating β_{lift} into the CPS transformation is achieved by refining the following rule in Definition 1:

$$\mathcal{E}[\![E[v_0\; v_1]]\!]\; k \quad = \quad \mathcal{V}[\![v_0]\!]\; \mathcal{V}[\![v_1]\!]\; (\mathcal{C}[\![E]\!]\; k)$$

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

$$\begin{array}{rcl} \mathcal{E}[\![E[x\; v_1]]\!]\; k & = & x\; \mathcal{V}[\![v_1]\!]\; (\mathcal{C}[\![E]\!]\; k) \\ \mathcal{E}[\![E[(\lambda x.e_0)\; v_1]]\!]\; k & = & (\lambda x.\mathcal{E}[\![E[e_0]]\!]\; k)\; \mathcal{V}[\![v_1]\!] \end{array}$$

As in Section 2, the refined context-based CPS transformation can be refocused to operate in one-pass and Church-encoded to be higher-order. Making it compositional, however, makes the CPS transformation dependently typed [12]. 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 eta-reductions and generalized reduction

Putting everything together, Definition 1 can be made tail-conscious and extended with administrative eta-reductions and generalized reduction. The result, if it is à la Fischer, coincides with Sabry and Felleisen's compacting CPS transformation [46, Definition 5]. It can be refocused to operate in one-pass and Church-encoded to be higher-order. But as in Section 5, making it compositional makes the CPS transformation dependently typed [12]. 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 syntactic theories, 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 [1, 21, 40] account for tail-consciousness, none pays a particular attention to administrative eta-reductions and to generalized reduction. For example, the context-based CPS transformation of the second edition of *Essentials of Programming Languages* [21] 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 [5, 19, 27, 34].

Acknowledgments: Thanks are due to Julia L. Lawall for comments.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[8] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, pages 209–220, Montréal, Canada, September 2000. ACM Press. Extended version to appear in the Journal of Functional Programming.

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

[10] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

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

[12] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. In Sabry [45], pages 35–39. Also available as the technical report BRICS RS-00-35.

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

[14] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001. Extended version available as the technical report BRICS RS-01-31.

[15] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. In Mogens Nielsen, editor, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, Lecture Notes in Computer Science, Grenoble, France, April 2002. Springer-Verlag. Extended version available as the technical report BRICS RS-01-49.

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

[17] Andrzej Filinski. An extensional CPS transform (preliminary report). In Sabry [45], pages 41–46.

[18] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. Earlier version available in the proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[19] 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.

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

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

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

[23] 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.

[24] Philippe de Groote. A CPS-translation of the $\lambda\mu$-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.

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

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

[27] 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.

[28] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303–319, 1997. Extended version available as the technical report BRICS RS-97-7.

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

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

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

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

[33] 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, June 1985. Springer-Verlag.

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

[35] 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.

[36] 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.

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

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

[39] 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.

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

[41] John Reppy. Local CPS conversion in a direct-style compiler. In Sabry [45], pages 1–6.

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

[43] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[44] 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.

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

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

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

[48] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974).

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

[50] 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.

[51] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of decomposition lemma. *Higher-Order and Symbolic Computation*, 14(4), 2001. To appear.

[52] Steve Zdancewic and Andrew Myers. Secure information flow and CPS. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 46–61, Genova, Italy, April 2001. Springer-Verlag.

Recent BRICS Report Series Publications

RS-02-3 **Olivier Danvy and Lasse R. Nielsen.** ***On One-Pass CPS Transformations*****. January 2002. 18 pp.**

RS-02-2 **Lasse R. Nielsen.** ***A Simple Correctness Proof of the Direct-Style Transformation*****. January 2002.**

RS-02-1 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***The*** **`<bigwig>`** ***Project*****. January 2002. 36 pp. This revised report supersedes the earlier BRICS report RS-00-42.**

RS-01-55 **Daniel Damian and Olivier Danvy.** ***A Simple CPS Transformation of Control-Flow Information*****. December 2001.**

RS-01-54 **Daniel Damian and Olivier Danvy.** ***Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation*****. December 2001. To appear in the** ***Journal of Functional Programming*****. This report supersedes the earlier BRICS report RS-00-15.**

RS-01-53 **Zoltán Ésik and Masami Ito.** ***Temporal Logic with Cyclic Counting and the Degree of Aperiodicity of Finite Automata*****. December 2001. 31 pp.**

RS-01-52 **Jens Groth.** ***Extracting Witnesses from Proofs of Knowledge in the Random Oracle Model*****. December 2001. 23 pp.**

RS-01-51 **Ulrich Kohlenbach.** ***On Weak Markov's Principle*****. December 2001. 10 pp.**

RS-01-50 **Jiří Srba.** ***Note on the Tableau Technique for Commutative Transition Systems*****. December 2001. 19 pp. To appear in the proceedings of FOSSACS '02.**

RS-01-49 **Olivier Danvy and Lasse R. Nielsen.** ***A First-Order One-Pass CPS Transformation*****. December 2001. 21 pp. Extended version of a paper to appear in the proceedings of FOSSACS '02.**

RS-01-48 **Mogens Nielsen and Frank D. Valencia.** ***Temporal Concurrent Constraint Programming: Applications and Behavior*****. December 2001. 36 pp.**

RS-01-47 **Jesper Buus Nielsen.** ***Non-Committing Encryption is Too Easy in the Random Oracle Model*****. December 2001. 20 pp.**

RS-01-46 **Lars Kristiansen.** ***The Implicit Computational Complexity of Imperative Programming Languages*****. November 2001. 46 pp.**