Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®
Theoretical Computer Science 308 (2003) 239–257
Theoretical Computer Science
www.elsevier.com/locate/tcs

A first-order one-pass CPS transformation☆

Olivier Danvy*,[1], Lasse R. Nielsen

BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark

Received 18 January 2002; received in revised form 26 August 2002; accepted 13 September 2002
Communicated by G.D. Plotkin

Abstract

We present a new transformation of λ-terms into continuation-passing style (CPS). This transformation operates in one pass and is both compositional and first-order. Previous CPS transformations only enjoyed two out of the three properties of being first-order, one-pass, and compositional, but the new transformation enjoys all three properties. It is proved correct directly by structural induction over source terms instead of indirectly with a colon translation, as in Plotkin's original proof. Similarly, it makes it possible to reason about CPS-transformed terms by structural induction over source terms, directly.

The new CPS transformation connects separately published approaches to the CPS transformation. It has already been used to state a new and simpler correctness proof of a direct-style transformation, and to develop a new and simpler CPS transformation of control-flow information.

Keywords: λ-calculus; Continuations; Continuation-passing style (CPS); Administrative reductions

1. Introduction

The transformation into continuation-passing style (CPS) is an encoding of arbitrary λ-terms into an evaluation-order-independent subset of the λ-calculus [31,37]. As already reviewed by Reynolds [36], continuations and the CPS transformation share a long history. The CPS transformation was first formalized by Plotkin [31], and first used

☆ A preliminary version is available in the proceedings of FOSSACS'02, Grenoble, France, April 2002.

* Corresponding author.
E-mail addresses: danvy@brics.dk (O. Danvy), lrn@brics.dk (L.R. Nielsen).

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

0304-3975/03/$ - see front matter
PII: S0304-3975(02)00733-8

in practice by Steele, in the first compiler for the Scheme programming language [41]. Unfortunately, its direct implementation as a rewriting system yields extraneous redexes known as *administrative redexes*. For example, the term $\lambda x.xx$ is CPS-transformed into

$$\lambda k.k\ (\lambda x.\lambda k.(\lambda k.k\,x)\ (\lambda x_0.(\lambda k.k\,x)\ (\lambda x_1.x_0 x_1\,k))).$$

The CPS-transformed term contains two administrative redexes: the two applications of $\lambda k.k\,x$. Reducing them yields two more administrative redexes:

$$\lambda k.k\,(\lambda x.\lambda k.(\lambda x_0.((\lambda x_1.x_0\,x_1\,k)\,x)\,x).$$

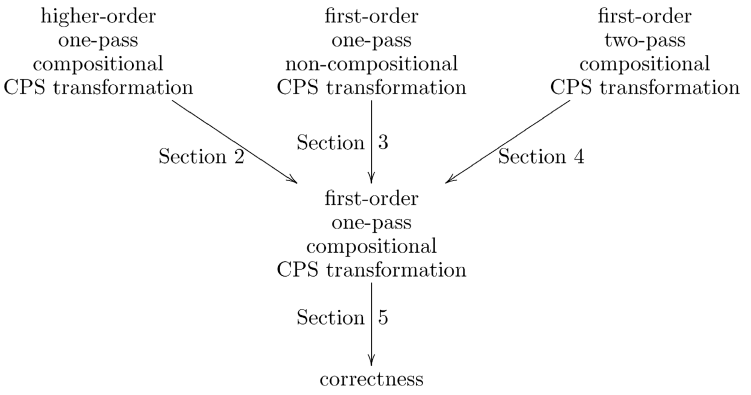
Administrative redexes interfere both with proving the correctness of a CPS transformation [31] and with using it in a compiler [23,41]. At the turn of the 1990s, two flavors of "one-pass" CPS transformations that contract administrative redexes at transformation time were developed. One flavor is compositional and higher-order, using a functional accumulator [1,10,43]. The other is non-compositional and first-order, using evaluation contexts [39]. They have both been proved correct and are used in compilers as well as to reason about CPS programs. Getting back to the example above, these one-pass CPS transformations directly map the term $\lambda x.xx$ into

$$\lambda k.k\ (\lambda x.\lambda k.x\,x\,k),$$

which contains no administrative redexes.

Because the existing one-pass CPS transformations are either higher-order or non-compositional, their correctness proofs are complicated, and so is reasoning about CPS-transformed programs. In this article, we present a one-pass CPS transformation that is both compositional and first-order and therefore is simple to prove correct and to reason about. It is also more efficient in practice.

Overview: The rest of this article is structured as follows. We present three derivations of our first-order, one-pass, and compositional CPS transformation. We derive it from the higher-order one-pass CPS transformation (Section 2), from Sabry and Wadler's non-compositional CPS transformation (Section 3), and from Steele's two-pass CPS transformation (Section 4). We also prove its correctness with a simulation theorem à la Plotkin (Section 5).



We then analyze the process of reasoning about CPS-transformed programs, depending on which kind of CPS transformation is used (Section 6). Finally, we conclude (Section 7).

Prerequisites: The syntax of the λ-calculus is as follows. We follow the tradition of distinguishing between trivial and serious terms. This distinction originates in Reynolds's work [37] and has been used by Moggi to distinguish between values and computations [25].

$$\begin{array}{lll} e ::= t \mid s & e \in \mathrm{Exp} & \text{(terms)},\\ t ::= x \mid \lambda x.e & t, K \in \mathrm{Val} & \text{(trivial terms, i.e. values)},\\ s ::= e_0\, e_1 & s \in \mathrm{Comp} & \text{(serious terms, i.e. computations)},\\ & x, k \in \mathrm{Ide} & \text{(identifiers)}. \end{array}$$

We identify terms modulo α-equivalence, i.e. modulo renaming of bound variables.

2. From higher-order to first-order

We start from a one-pass, compositional, and higher-order CPS transformation and we make it first order.

2.1. A higher-order specification

Fig. 1 displays a higher-order, one-pass, compositional CPS transformation. Transformation-time abstractions ($\bar{\lambda}$) and applications (infix $\bar{@}$) are overlined. Underlined abstractions ($\underline{\lambda}$) and applications (infix $\underline{@}$) are hygienic syntax constructors, i.e. they generate terms with fresh variables.

The transformation is defined with five functions:

- $\mathscr{E}$ is applied to terms in tail position [3] and $\mathscr{E}'$ to terms appearing in non-tail position; they are otherwise similar.
- $\mathscr{S}$ is applied to serious terms in tail position and $\mathscr{S}'$ to terms appearing in non-tail position; they are otherwise similar.
- $\mathscr{T}$ is applied to trivial terms.

A term e is CPS-transformed into the result of reducing the transformation-time abstractions and applications in $\underline{\lambda} k.\mathscr{E}[\![e]\!]\bar{@} k$.

2.2. Circumventing the functional accumulator

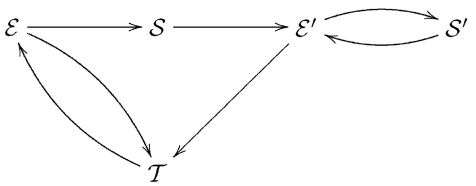
Let us analyze the function spaces in Fig. 1. All the calls to $\mathscr{E}$, $\mathscr{S}$, $\mathscr{E}'$, and $\mathscr{S}'$ are fully applied and thus these functions could as well be uncurried. The resulting CPS transformation is only higher-order because of the functional accumulator used in $\mathscr{E}'$ and $\mathscr{S}'$. Let us circumvent it.

A simple control-flow analysis of the uncurried CPS transformation tells us that while both $\mathscr{E}$ and $\mathscr{E}'$ invoke $\mathscr{T}, \mathscr{T}$ only invokes $\mathscr{E}$, $\mathscr{E}$ only invokes $\mathscr{S}$, and $\mathscr{S}$ only invokes $\mathscr{E}'$ while $\mathscr{E}'$ and $\mathscr{S}'$ invoke each other. The following diagram illustrates these

$$
\begin{aligned}
&\mathscr{E} : \mathrm{Expr} \to \mathrm{Ide} \to \mathrm{Comp}\\
&\quad \mathscr{E}[\![t]\!] = \bar{\lambda} k.k \mathbin{\underline{@}} \mathscr{T}[\![t]\!]\\
&\quad \mathscr{E}[\![s]\!] = \bar{\lambda} k.\mathscr{S}[\![s]\!] \bar{@} k\\
&\mathscr{S} : \mathrm{Comp} \to \mathrm{Ide} \to \mathrm{Comp}\\
&\quad \mathscr{S}[\![e_0\ e_1]\!] = \bar{\lambda} k.\mathscr{E}'[\![e_0]\!] \bar{@} (\bar{\lambda} x_0.\mathscr{E}'[\![e_1]\!] \bar{@} (\bar{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k))\\
&\mathscr{T} : \mathrm{Val} \to \mathrm{Val}\\
&\quad \mathscr{T}[\![x]\!] = x\\
&\quad \mathscr{T}[\![\lambda x.e]\!] = \underline{\lambda} x.\underline{\lambda} k.\mathscr{E}[\![e]\!] \bar{@} k\\
&\mathscr{E}' : \mathrm{Expr} \to (\mathrm{Val} \to \mathrm{Comp}) \to \mathrm{Comp}\\
&\quad \mathscr{E}'[\![t]\!] = \bar{\lambda} \kappa.\kappa \bar{@} \mathscr{T}[\![t]\!]\\
&\quad \mathscr{E}'[\![s]\!] = \bar{\lambda} \kappa.\mathscr{S}'[\![s]\!] \bar{@} \kappa\\
&\mathscr{S}' : \mathrm{Comp} \to (\mathrm{Val} \to \mathrm{Comp}) \to \mathrm{Comp}\\
&\quad \mathscr{S}'[\![e_0\ e_1]\!] = \bar{\lambda} \kappa.\mathscr{E}'[\![e_0]\!] \bar{@} (\bar{\lambda} x_0.\mathscr{E}'[\![e_1]\!] \bar{@} (\bar{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} (\underline{\lambda} x_2.\kappa \bar{@} x_2)))
\end{aligned}
$$

Fig. 1. Higher-order one-pass CPS transformation.

relationships.



Therefore, if we could prevent $\mathscr{S}$ from calling $\mathscr{E}'$, both $\mathscr{E}'$ and $\mathscr{S}'$ would become dead code, and only $\mathscr{E}$, $\mathscr{S}$, and $\mathscr{T}$ would remain.

Let us unfold the definition of $\mathscr{S}$ and reason by inversion, i.e. by enumerating all the possibilities. The four following cases occur. (We only detail the transformation-time β-reductions in the first case.)

$$
\begin{aligned}
\mathscr{S}[\![t_0\,t_1]\!] \bar{@} k &=_{\mathrm{def}} \mathscr{E}'[\![t_0]\!] \bar{@} (\bar{\lambda} x_0.\mathscr{E}'[\![t_1]\!] \bar{@} (\bar{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k))\\
&=_{\mathrm{def}} (\bar{\lambda} x_0.\mathscr{E}'[\![t_1]\!] \bar{@} (\bar{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k)) \bar{@} \mathscr{T}[\![t_0]\!]\\
&\to_{\beta} \mathscr{E}'[\![t_1]\!] \bar{@} (\bar{\lambda} x_1.\mathscr{T}[\![t_0]\!] \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k)\\
&=_{\mathrm{def}} (\bar{\lambda} x_1.\mathscr{T}[\![t_0]\!] \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k) \bar{@} \mathscr{T}[\![t_1]\!]\\
&\to_{\beta} \mathscr{T}[\![t_0]\!] \mathbin{\underline{@}} \mathscr{T}[\![t_1]\!] \mathbin{\underline{@}} k,
\end{aligned}
$$

$$\mathscr{S}[\![t_0\,s_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}'[\![s_1]\!]\,\bar{@}\,(\bar{\lambda}x_1.\mathscr{T}[\![t_0]\!]\ \underline{@}\ x_1\ \underline{@}\ k),$$

$$\mathscr{S}[\![s_0\,t_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}'[\![s_0]\!]\,\bar{@}\,(\bar{\lambda}x_0.x_0\ \underline{@}\ \mathscr{T}[\![t_1]\!]\ \underline{@}\ k),$$

$$\mathscr{S}[\![s_0\,s_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}'[\![s_0]\!]\,\bar{@}\,(\bar{\lambda}x_0.\mathscr{S}'[\![s_0]\!]\,\bar{@}\,(\bar{\lambda}x_1.x_0\ \underline{@}\ x_1\ \underline{@}\ k)).$$

This analysis makes explicit all of the functions κ that $\mathscr{S}$ passes to $\mathscr{S}'$. By definition of $\mathscr{S}'$, we also know *where* these functions are applied: in the two-level eta-redex $\underline{\lambda}\,x_2.\kappa\,\bar{@}\,x_2$. We can take advantage of this knowledge by invoking $\mathscr{S}$ rather than $\mathscr{S}'$, extend its domain to $\mathrm{Comp} \to \mathrm{Exp} \to \mathrm{Comp}$, and pass it the result of eta-expanding κ. The result reads as follows:

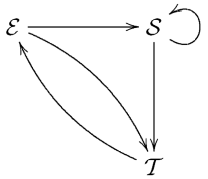
$$\mathscr{S}[\![t_0\,t_1]\!]\,\bar{@}\,k =_\beta \mathscr{T}[\![t_0]\!]\ \underline{@}\ \mathscr{T}[\![t_1]\!]\ \underline{@}\ k,$$

$$\mathscr{S}[\![t_0\,s_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}[\![s_1]\!]\,\bar{@}\,(\underline{\lambda}\,x_1.\mathscr{T}[\![t_0]\!]\ \underline{@}\ x_1\ \underline{@}\ k),$$

$$\mathscr{S}[\![s_0\,t_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}[\![s_0]\!]\,\bar{@}\,(\underline{\lambda}\,x_0.x_0\ \underline{@}\ \mathscr{T}[\![t_1]\!]\ \underline{@}\ k),$$

$$\mathscr{S}[\![s_0\,s_1]\!]\,\bar{@}\,k =_\beta \mathscr{S}[\![s_0]\!]\,\bar{@}\,(\underline{\lambda}\,x_0.\mathscr{S}[\![s_1]\!]\,\bar{@}\,(\underline{\lambda}\,x_1.x_0\ \underline{@}\ x_1\ \underline{@}\ k)).$$

In this derived transformation, $\mathscr{E}'$ and $\mathscr{S}'$ are no longer used. Since they are the only higher-order components of the uncurried CPS transformation, the derived transformation, while still one-pass and compositional, is first-order. Its control-flow graph can be depicted as follows:



The resulting CPS transformation is shown in Fig. 2. Since it is first-order, there are no overlined abstractions and applications—all abstractions and applications are underlined. Therefore, for notational simplicity, we omit all underlines as well as the infix @. A term e is CPS-transformed into $\lambda k.\mathscr{E}[\![e]\!]k$.

This first-order CPS transformation is compositional (in the sense of denotational semantics) because on the right-hand side, all recursive calls are on proper sub-parts of the left-hand side term [44, p. 60]. One could say, however, that it is not purely defined by recursive descent, since $\mathscr{S}$ is defined by cases on immediate sub-terms, using a sort of structural look-ahead. (A change of grammar would solve that problem, though.) The main cost incurred by the inversion step above is that in general it requires 2^n clauses for a source term with n sub-terms that need to be considered (e.g., a tuple).

3. From non-compositional to compositional

We start from a one-pass, first order, and non-compositional CPS transformation and we make it compositional.

$$
\begin{aligned}
&\mathscr{E} : \mathrm{Expr} \times \mathrm{Ide} \to \mathrm{Comp}\\
&\qquad \mathscr{E}[\![t]\!]k = k\ \mathscr{T}[\![t]\!]\\
&\qquad \mathscr{E}[\![s]\!]k = \mathscr{S}[\![s]\!]k\\
&\\
&\mathscr{S} : \mathrm{Comp} \times \mathrm{Expr} \to \mathrm{Comp}\\
&\qquad \mathscr{S}[\![t_0\ t_1]\!]K = \mathscr{T}[\![t_0]\!]\ \mathscr{T}[\![t_1]\!]\ K\\
&\qquad \mathscr{S}[\![t_0\ s_1]\!]K = \mathscr{S}[\![s_1]\!](\lambda x_1.\mathscr{T}[\![t_0]\!]\ x_1\ K)\\
&\qquad \mathscr{S}[\![s_0\ t_1]\!]K = \mathscr{S}[\![s_0]\!](\lambda x_0.x_0\ \mathscr{T}[\![t_1]\!]\ K)\\
&\qquad \mathscr{S}[\![s_0\ s_1]\!]K = \mathscr{S}[\![s_0]\!](\lambda x_0.\mathscr{S}[\![s_1]\!](\lambda x_1.x_0\ x_1\ K))\\
&\\
&\mathscr{T} : \mathrm{Val} \to \mathrm{Val}\\
&\qquad \mathscr{T}[\![x]\!] = x\\
&\qquad \mathscr{T}[\![\lambda x.e]\!] = \lambda x.\lambda k.\mathscr{E}[\![e]\!]k
\end{aligned}
$$

Fig. 2. First-order one-pass CPS transformation.

3.1. A non-compositional specification

The first edition of *Essentials of Programming Languages* [18] dedicated a chapter to the CPS transformation, with the goal to be as intuitive and pedagogical as possible and to produce CPS terms similar to what one would write by hand. This CPS transformation inspired Sabry and Felleisen to design a radically different CPS transformation based on evaluation contexts that produces a remarkably compact output due to an extra reduction rule, β_{lift} [11,39]. Sabry and Wadler then simplified this CPS transformation [40, Fig. 18], e.g., by omitting β_{lift}. This simplified CPS transformation now forms the basis of the chapter on the CPS transformation in the second edition of *Essentials of Programming Languages* [19].

Using the same notation as in Fig. 2, Sabry and Wadler's CPS transformation reads as follows. A term e is CPS-transformed into $\lambda k.\mathscr{E}[\![e]\!]$, where

$$
\begin{aligned}
\mathscr{E}[\![e]\!] &= \mathscr{S}[\![e]\!]k,\\
\mathscr{S}[\![t]\!]K &= K\ \mathscr{T}[\![t]\!],\\
\mathscr{S}[\![t_0\ t_1]\!]K &= \mathscr{T}[\![t_0]\!]\ \mathscr{T}[\![t_1]\!]K,\\
\mathscr{S}[\![t_0\ s_1]\!]K &= \mathscr{T}[\![s_1]\!]\ (\lambda x_1.\mathscr{S}[\![t_0\ x_1]\!]K),\\
\mathscr{S}[\![s_0\ e_1]\!]K &= \mathscr{S}[\![s_0]\!]\ (\lambda x_0.\mathscr{S}[\![x_0\ e_1]\!]K),\\
\mathscr{T}[\![x]\!] &= x,\\
\mathscr{T}[\![\lambda x.e]\!] &= \lambda x.\lambda k.\mathscr{E}[\![e]\!].
\end{aligned}
$$

For each serious term s with a serious immediate sub-term s', $\mathscr{S}$ recursively traverses s' with a new continuation. In this new continuation, s' is replaced by a fresh vari-

able (i.e. a trivial immediate sub-term) in s. The result, now with one less serious immediate sub-term, is transformed recursively. The idea was the same in Sabry and Felleisen's context-based CPS transformation [39, Definition 5], which we study elsewhere [12,14,27].

These CPS transformations hinge on a unique free variable k and also they are not compositional. For example, on the right-hand side of the definition of $\mathscr{S}$ just above, some recursive calls are on terms that are not proper sub-parts of the left-hand side term. The input program changes dynamically during the transformation, and correspondingly, termination is proved using a size argument [39, Definition 6]. In contrast, a compositional transformation entails a simpler termination proof by structural induction.

3.2. Eliminating the non-compositionality

Sabry and Wadler's CPS transformation can be made compositional through the following unfolding steps.

Unfolding $\mathscr{S}$ in $\mathscr{S}[\![t_0\, x_1]\!]K$: The result is $\mathscr{T}[\![t_0]\!]\mathscr{T}[\![x_1]\!]K$, which is equivalent to $\mathscr{T}[\![t_0]\!]\, x_1 K$.

Unfolding $\mathscr{S}$ in $\mathscr{S}[\![x_0\, e_1]\!]K$: Two cases occur (thus splitting this clause for $\mathscr{S}$ into two).

- If e_1 is a value (call it t_1), the result is $\mathscr{T}[\![x_0]\!]\mathscr{T}[\![t_1]\!]K$, which is equivalent to $\mathscr{T}[\![t_1]\!]K$.
- If e_1 is a computation (call it s_1), the result is $\mathscr{S}[\![s_1]\!]\ (\lambda x_1.\mathscr{S}[\![x_0\, x_1]\!]K)$. Unfolding the inner occurrence of $\mathscr{S}$ yields $\mathscr{S}[\![s_1]\!]\ (\lambda x_1.\mathscr{T}[\![x_0]\!]\mathscr{T}[\![x_1]\!]K)$, which is equivalent to $\mathscr{S}[\![s_1]\!](\lambda x_1.x_0\, x_1\, K)$.

The resulting unfolded transformation is compositional. It also coincides with the definition of $\mathscr{S}$ in Fig. 2 and thus connects the two approaches to the CPS transformation that have been separately reported in the literature.

4. From two passes to one pass

We start from a first-order, compositional, and two-pass CPS transformation and we make it operate in one pass.

4.1. A two-pass specification

Plotkin's CPS transformation [31] can be phrased as follows.

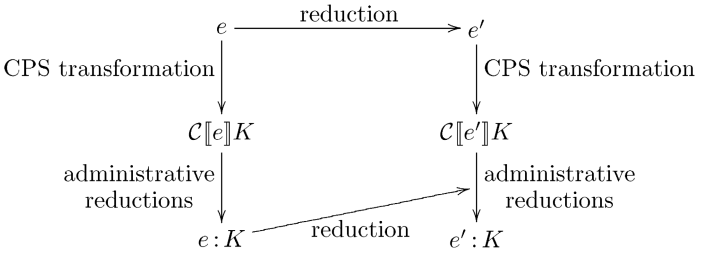
$$\begin{aligned}
\mathscr{C}[\![t]\!] &= \lambda k.k\ \Phi(t)\\
\mathscr{C}[\![e_0\, e_1]\!] &= \lambda k.\mathscr{C}[\![e_0]\!]\ (\lambda x_0.\mathscr{C}[\![e_1]\!]\ (\lambda x_1.x_0\, x_1 k))\\
\Phi(x) &= x\\
\Phi(\lambda x.e) &= \lambda x.\mathscr{C}[\![e]\!]
\end{aligned}$$

As illustrated in Section 1, a direct implementation of this transformation yields CPS terms containing a mass of administrative redexes that need to be contracted in a second pass [41].

4.2. A colon translation for proving simulation

Plotkin's simulation theorem shows a correspondence between reductions in the source program and in the transformed program. To this end, he introduced the so-called "colon translation" to bypass the initial administrative reductions of a CPS-transformed term.

The colon translation makes it possible to focus on the reduction of the abstractions inherited from the source program. The simulation theorem is shown by relating each reduction step, as depicted by the following diagram:



The bottom arrow points to the middle of another arrow to express that reduction in general yields a term that results from the administrative reduction of a CPS-transformed term and that can be further administratively reduced.

The colon translation is itself a CPS transformation. It transforms a source term and a continuation into a CPS term; this CPS term is the one that appears after contracting the initial administrative redexes of the CPS-transformed term applied to the continuation. In other words, if we write the colon translation of the term e and the continuation K as $e : K$, then the following holds: $\mathscr{C}[\![e]\!]K \xrightarrow{*} e : K$.

The colon translation can be derived from the CPS transformation by predicting the result of the initial administrative reductions from the structure of the source term. For example, a serious term of the form $t_0\, e_1$ is CPS-transformed into $\lambda k.(\lambda k.k\, \Phi(t_0))$ $(\lambda x_0.\mathscr{C}[\![e_1]\!]\, (\lambda x_1.x_0\, x_1\, k))$. Applying this CPS term to a continuation enables the following administrative reductions:

$$
\begin{aligned}
&(\lambda k.(\lambda k.k\ \Phi(t_0))\ (\lambda x_0.\mathscr{C}[\![e_1]\!]\ (\lambda x_1.x_0\, x_1\, k)))\, K\\
&\to_\beta (\lambda k.k\ \Phi(t_0))\ (\lambda x_0.\mathscr{C}[\![e_1]\!]\ (\lambda x_1.x_0\, x_1\, K))\\
&\to_\beta (\lambda x_0.\mathscr{C}[\![e_1]\!](\lambda x_1.x_0\, x_1\, K))\Phi(t_0)\\
&\to_\beta \mathscr{C}[\![e_1]\!]\, \lambda x_1.\Phi(t_0)\, x_1\, K.
\end{aligned}
$$

The result is a smaller term that can be CPS-transformed recursively. This insight leads one to Plotkin's colon translation, as follows:

$$
\begin{aligned}
t : K &= K\, \Phi(t),\\
t_0\, t_1 : K &= \Phi(t_0)\, \Phi(t_1)\, K,\\
t_0\, s_1 : K &= s_1 : (\lambda x_1.\Phi(t_0)\, x_1\, K),\\
s_0\, e_1 : K &= s_0 : (\lambda x_0.\mathscr{C}[\![e_1]\!](\lambda x_1.x_0\, x_1\, K)).
\end{aligned}
$$

4.3. Merging CPS transformation and colon translation

For Plotkin's purpose—reasoning about the output of the CPS transformation—contracting the initial administrative reductions in each step is sufficient. Our goal, however, is to remove all administrative redexes in one pass. Since the colon translation contracts some administrative redexes, and thus more than the CPS transformation alone, further administrative redexes can be contracted by using the colon translation in place of all occurrences of $\mathscr{C}$.

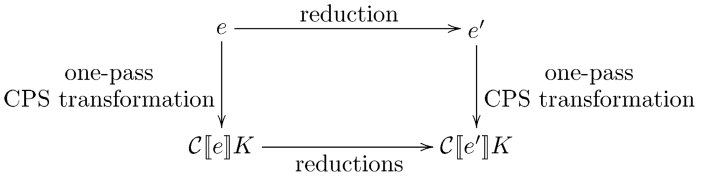
The CPS transformation is used once in the colon translation and once in the definition of Φ. For consistency, we distinguish two cases in the colon translation, depending on whether the term is a value or not, and we use the colon translation if it is not a value. In the definition of Φ, we introduce the continuation identifier and then we use the colon translation. The resulting extended colon translation reads as follows:

$$\begin{aligned}
t : K &= K\,\Phi(t),\\
t_0\,t_1 : K &= \Phi(t_0)\,\Phi(t_1)\,K,\\
t_0\,s_1 : K &= s_1 : (\lambda x_1.\Phi(t_0)\,x_1\,K),\\
s_0\,t_1 : K &= s_0 : (\lambda x_0.x_0\,\Phi(t_1)\,K),\\
s_0\,s_1 : K &= s_0 : (\lambda x_0.(s_1 : (\lambda x_1.x_0\,x_1\,K))),\\
\Phi(x) &= x,\\
\Phi(\lambda x.e) &= \lambda x.\lambda k.(e : k).
\end{aligned}$$

With a change of notation, this extended colon translation coincides with the first-order one-pass CPS transformation from Fig. 2. In other words, not only does the extended colon translation remove more administrative redexes than the original one, but it actually removes as many as the two-pass transformation.

5. A direct proof of Plotkin's simulation theorem

Plotkin established the correctness proof of the CPS transformation with a simulation theorem [31]. His proof was indirect in that he used a so-called colon translation to account for the administrative redexes—a proof technique that has been repeatedly used in the subsequent literature [20,22,29,38]. [2] In contrast, we present a direct proof of the simulation theorem for the CPS transformation of Fig. 2. This simulation theorem is shown by relating each reduction step, as depicted by the following diagram:



But first, we formally define a reduction relation on programs.

[2] More often, though, administrative redexes are left out in published work [26,45].

5.1. Reduction rules

We give the reduction relation using evaluation contexts in the style of Felleisen [17]. The evaluation contexts are given by the following grammar:

$$E ::= [] \mid E\,e \mid t\,E.$$

A context is a term with a hole. We plug the hole of a context E with a term e (denoted $E[e]$) as follows:

$$\begin{aligned} [][e] &= e, \\ (E\,e')[e] &= (E[e])\,e', \\ (t\,E)[e] &= t\,(E[e]). \end{aligned}$$

This definition of evaluation contexts satisfies a unique decomposition property, namely that any term that is not a value can be decomposed into a context and an application of values, i.e.

$$\forall s.\exists E, t_0, t_1.s = E[t_0\,t_1]$$

and this decomposition is unique. Since decomposition is unique, a term can match a reduction rule in at most one way. Therefore the reduction relation is deterministic.

We then define a reduction relation on terms with the following rule:

$$E[(\lambda x.e)t] \to E[e[t/x]]$$

where $e[t/x]$ is the usual capture-avoiding substitution of t for free occurrences of x in e. A term of the form $(\lambda x.e)t$ is a *redex*.

We say that e is reducible if there exists an e' such that $e \to e'$. Only the terms of the form $E[(\lambda x.e)t]$ are reducible. We write $\overset{+}{\to}$, $\overset{*}{\to}$, and $\overset{n}{\to}$ for the transitive closure, the reflexive and transitive closure, and the n-times composition of the relation $\to$.

Some computations are not reducible. They are said to be stuck. The set of stuck terms is exactly those of the form $E[xt]$, i.e. the application of a *variable* to a value in an evaluation context. Since decomposition is unique, such a term cannot be reducible.

A sequence of reductions $e_0 \to e_1 \to \cdot \to e_n$ is simply called a *reduction sequence* (of e_0). A maximal reduction sequence, i.e. either an infinite one or one ending in a value or a stuck term, is called a *derivation* (of e_0). Since reduction is deterministic, there is exactly one derivation of any term.

5.2. A simulation theorem

Plotkin used four lemmas and a colon translation to prove the correctness of his CPS transformation. Since our CPS transformation already performs the administrative reductions at transformation time, we do not need to introduce any colon translation and thus Plotkin's initial-reduction lemma holds trivially. Therefore, we work directly with the CPS transformation in the following three lemmas:

Lemma 1 (Substitution). *If e is a term, t a value, x a variable, and K another value then*

$$(\mathscr{E}[\![e]\!]K)[\mathscr{T}[\![t]\!]/x] = \mathscr{E}[\![e[t/x]]\!]\ (K[\mathscr{T}[\![t]\!]/x]).$$

If e is a term, k is a variable, and K a term then

$$(\mathscr{E}[\![e]\!]k)[K/k] = \mathscr{E}[\![e]\!]K.$$

Proof. The first equation is proved by induction on the structure of e, following the definition of substitution.

The second equation follows directly from the definition of $\mathscr{E}[\![e]\!]K$. □

Lemma 2 (Single-step simulation). *The reductions of the transformed program match the reductions of the source program in the sense that*

$$e \to e' \Rightarrow \mathscr{E}[\![e]\!]K \xrightarrow{+} \mathscr{E}[\![e']\!]K.$$

Proof. If $e \to e'$ then there exists a context E, a redex $t_0\,t_1$, and a term e'' such that $e = E[t_0\,t_1]$ and $e' = E[e'']$. The proof, which we omit, is by induction on the context E. □

Lemma 2 accounts for all reducible terms. The following lemma handles the terms that are stuck.

Lemma 3 (Preservation of stuck terms). *If e is stuck* (*i.e., if it is a computation that is not reducible*) *and K is a value, then* $\mathscr{S}[\![e]\!]K$ *is stuck.*

Proof. Since all stuck terms are of the form $E[xt]$, the proof is by induction on E. □

Theorem 1 (Simulation). $\forall e, v.\,(\exists t.e \xrightarrow{*} t \wedge \mathscr{T}[\![t]\!] = v) \;\Leftrightarrow\; \mathscr{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v.$

Proof. Let e and v be given. We consider in turn the two directions of the biimplication.

1. $(\exists t.e \xrightarrow{*} t \wedge \mathscr{T}[\![t]\!]=v) \Rightarrow \mathscr{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v$. We assume $(\exists t.e \xrightarrow{*} t \wedge \mathscr{T}[\![t]\!]=v)$ and we choose a t such that $e \xrightarrow{*} t$ and $v = \mathscr{T}[\![t]\!]$.

   From repeated use of Lemma 2, it follows that $\mathscr{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} \mathscr{E}[\![t]\!]\,\lambda x.x$, and $\mathscr{E}[\![t]\!]\,\lambda x.x = (\lambda x.x)\mathscr{T}[\![t]\!] \to \mathscr{T}[\![t]\!] = v$.
2. $\mathscr{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v \Rightarrow (\exists t.e \xrightarrow{*} t \wedge \mathscr{T}[\![t]\!]=v)$. This implication is proved by contraposition, i.e. we assume that $\exists t.e \xrightarrow{*} t \wedge \mathscr{T}[\![t]\!] = v$ fails to hold. This failure can be due to there being no t such that $e \xrightarrow{*} t$ or to there being a t such that $e \xrightarrow{*} t$ but $\mathscr{T}[\![t]\!] \neq v$. Since the relation, $\to$, is deterministic, there can be at most one t such that $e \xrightarrow{*} t$.
   - We assume that there is no t such that $e \xrightarrow{*} t$.

This can happen in two scenarios: when e diverges, i.e. has an infinite derivation, and when it reduces to a stuck term. In either case $\mathscr{E}[\![e]\!]\lambda x.x$ has the same behavior.

- e diverges $\Rightarrow$ $\mathscr{E}[\![e]\!]\lambda x.x$ diverges. If a term e diverges, there exists no finite number n such that $e \xrightarrow{n} e'$ and e' is not reducible (i.e. either a value or a stuck term). That is, for all numbers n there exists a term e_n such that $e \xrightarrow{n} e_n$. Now, let n be an arbitrary natural number. We consider the sequence $e \to e_1 \to \cdots \to e_n$, which exists since e diverges. Then, from Lemma 2 we know that there is another reduction sequence $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{+} \mathscr{E}[\![e_1]\!]\lambda x.x \xrightarrow{+} \cdots \xrightarrow{+} \mathscr{E}[\![e_n]\!]\lambda x.x$ of length at least n. Therefore $\mathscr{E}[\![e]\!]\lambda x.x$ has reduction sequences of arbitrary length and thus it diverges as well.
  Since $\mathscr{E}[\![e]\!]\lambda x.x$ diverges, it cannot be the case that $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} v$.
- $e \xrightarrow{*} e'$ and e' is stuck $\Rightarrow$ $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} e''$ and e'' is stuck.
  From repeated use of Lemma 2 we know that $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} \mathscr{E}[\![e']\!]\lambda x.x$, and from Lemma 3 we know that $\mathscr{E}[\![e']\!]\lambda x.x$ is stuck.
  Since a stuck term cannot be a value, it cannot be the case that $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} v$.

- We assume that $e \xrightarrow{*} t$ but $\mathscr{T}[\![t]\!] \neq v$.
  From repeated use of Lemma 2 we know that $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} \mathscr{E}[\![t]\!]\lambda x.x$, and by definition $\mathscr{E}[\![t]\!]\lambda x.x \to \mathscr{T}[\![t]\!]$. Since $\mathscr{T}[\![t]\!] \neq v$, it cannot be the case that $\mathscr{E}[\![e]\!]\lambda x.x \xrightarrow{*} v$.

These cases prove the contraposition of the second implication.

Together these two directions prove the simulation theorem. □

6. Reasoning about CPS-transformed programs

How to go about proving properties of CPS-transformed programs depends on which kind of CPS transformation was used. In this section, we review each of them in turn, and we examine to which extent one can reason about CPS-transformed programs by structural induction over source programs (i.e. over programs prior to the CPS transformation).

We use two running examples: (1) proving that the CPS transformation yields well-formed CPS terms, i.e. terms that satisfy a BNF expressed, e.g., as an ML data type; and (2) proving that the CPS transformation preserves types [24,42]. (We consider the simply typed λ-calculus, with a typing judgment of the form $\Gamma \vdash e : \tau$.)

6.1. A higher-order one-pass CPS transformation

Danvy and Filinski used a typing argument to prove that their one-pass CPS transformation is well-defined [10, Theorem 1]. To prove the corresponding simulation theorem, they used a notion of schematic continuations. Since then, for the same purpose, we have developed a higher-order analogue of Plotkin's colon translation [13,27].

Proving structural properties of CPS programs is not completely trivial. Matching the higher-order nature of the one-pass CPS transformation, a logical relation is needed,

e.g., to prove ordering properties of CPS terms [9,15,16]. (The analogy between these ordering properties and substitution properties of linear λ-calculi has prompted Polakow and Pfenning to develop an ordered logical framework [32–34].) A logical relation amounts to structural induction at higher types. Therefore, it is crucial that the higher-order one-pass CPS transformation be compositional.

The CPS transformation yields well-formed terms: To prove well-formedness, it is enough to observe that the CPS transformation itself is a well-typed function from the data type of direct-style terms to the data type of CPS terms.

The CPS transformation preserves types: To prove well-typedness, we proceed by structural induction on the typing derivation of the source term (or by structural induction on the source term), together with a logical relation on the functional accumulator.

6.2. A first-order two-pass CPS transformation

Sabry and Felleisen also considered a two-pass CPS transformation. They used developments [2, Section 11.2] to prove that it is total [39, Proposition 2].

To prove structural properties of simplified CPS programs, one can (1) characterize the property prior to simplification, and (2) prove that simplifications preserve the property. Danvy took these steps to prove occurrence conditions of continuation identifiers [8], and so did Damian and Danvy to characterize the effect of the CPS transformation on control flow and binding times [4,6]. It is Polakow's thesis that an ordered logical framework provides a good support for stating and proving such properties [32,35].

The CPS transformation yields well-formed terms: To prove well-formedness, it is enough to observe that the first pass of the CPS transformation is a well-typed function from the data type of direct-style terms to the data type of CPS terms, and that the grammar of CPS terms is closed under administrative reduction.

The CPS transformation preserves types: To prove well-typedness, we first proceed by structural induction on the typing derivation of the source term. (It is thus crucial that the CPS transformation be compositional.) For the second pass, we need to show that the administrative contractions preserve the typeability and the type of the result. But this follows from the subject reduction property of the simply typed λ-calculus.

6.3. A first-order one-pass CPS transformation

The proof in Section 5 follows the spirit of Plotkin's original proof [31] but is more direct since it does not require a colon translation.

A first-order CPS transformation makes it possible to prove structural properties of a CPS-transformed program by structural induction on the source program. We find these proofs noticeably simpler than the ones mentioned in Section 6.1. For two other examples, Damian and Danvy have used the present first-order CPS transformation to develop a CPS transformation of control-flow information [5] that is simpler than existing ones [4,6,30], and Nielsen has used it to present a new and simpler correctness proof of a direct-style transformation [27,28].

Again, for structural induction to go through, it is crucial that the CPS transformation be compositional.

The CPS transformation yields well-formed terms: To prove well-formedness, it is enough to observe that the CPS transformation is a well-typed function from the data type of direct-style terms to the data type of CPS terms.

The CPS transformation preserves types: To prove well-typedness, we proceed by structural induction on the typing derivation of the source term.

6.4. Non-compositional CPS transformations

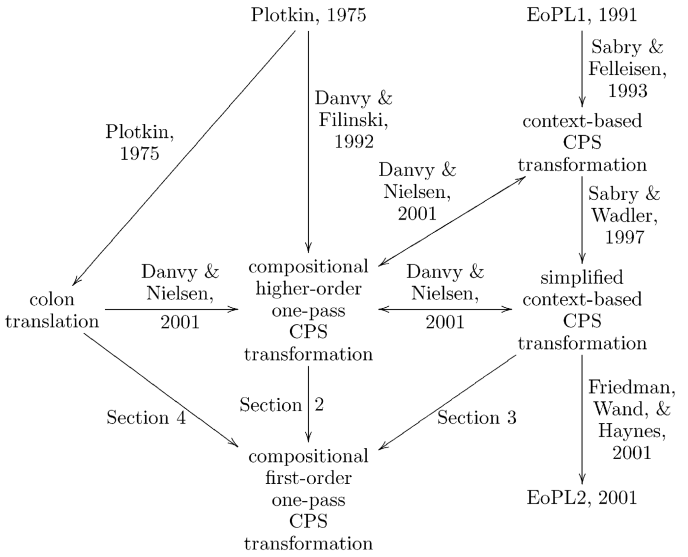
Sabry and Felleisen's proofs are by induction on the size of the source program [39, Appendix A, p. 337]. Proving that a CPS-transformed term is well formed and proving that it is well typed require a substitution lemma.

In their study of the computational λ-calculus λ_c, Sabry and Wadler specify a translation from λ_c to a simplified monadic calculus [40, Fig. 8]. This translation is non-compositional in the same sense that the CPS transformation of Section 3.1 is non-compositional. Sabry and Wadler, however, state that they proved properties of this translation (Parts (1) and (2) of Proposition 5.3) by structural induction over its input, which must have required unfolding steps similar to those of Section 3.2. In any case, their translation can be made compositional by following the same steps as in Section 3.2, which enables a direct proof by structural induction.

7. Conclusion and issues

7.1. The big picture

Elsewhere [11,12,14], we have developed further connections between higher-order and context-based one-pass CPS transformations. The overall situation is summarized in the following diagram:



This diagram is clearly in two parts: the left part stems from Plotkin's work and the right part from the first edition of *Essentials of Programming Languages*. The left-most part represents the CPS transformation with the colon translation. The vertical line in the middle represents the path of compositional CPS transformations. The vertical line on the right represents the path of non-compositional CPS transformations. The right arrow from the colon translation is our higher-order colon translation [13]. The upper arrows between the left part and the right part of the diagram correspond to our work on β-redexes [11], defunctionalization [12], and refocusing in syntactic theories [14].

The present work links the left part and the right part of the diagram further.

7.2. Scaling up

Our derivation of a first-order, one-pass CPS transformation generalizes to other evaluation orders, e.g., call-by-name. (Indeed each evaluation order gives rise to a different CPS transformation [21].) The CPS transformation also scales up to the usual syntactic constructs of a programming language such as primitive operations, tuples, conditional expressions, and sequencing.

A practical problem, however, arises for block structure, i.e. let- and letrec-expressions. For example, a let-expression is CPS-transformed as follows (extending Fig. 1).

$$\mathscr{S}[\![\text{let}\, x = e_1 \ \text{in} \ e_2]\!] = \bar{\lambda} k.\mathscr{E}[\![e_1]\!] \bar{@} (\underline{\lambda} x.\mathscr{E}[\![e_2]\!] \bar{@} k),$$

$$\mathscr{S}'[\![\text{let}\, x = e_1 \ \text{in} \ e_2]\!] = \bar{\lambda} \kappa.\mathscr{E}'[\![e_1]\!] \bar{@} (\underline{\lambda} x.\mathscr{E}'[\![e_2]\!] \bar{@} \kappa).$$

In contrast to Section 2.2, the call site of the functional accumulator (i.e. where it is applied) cannot be determined in one pass with finite look-ahead. This information is context sensitive because κ can be applied in arbitrarily deeply nested blocks. We do not see how a first-order one-pass CPS transformation can flatten nested blocks in general if it is also to be compositional.

To flatten nested blocks, one can revert to a non-compositional CPS transformation, to a two-pass CPS transformation, or to a higher-order CPS transformation. Elsewhere [11], we have shown that such a higher-order, compositional, and one-pass CPS transformation is dependently typed; its type depends on the nesting depth.

In the course of this work, and in the light of Section 3.2, we have conjectured that the problem of block structure should also apply to a first-order one-pass CPS transformation such as Sabry and Wadler's. This is the topic of the next section.

7.3. A shortcoming

Sabry and Wadler's transformation [40] also handles let expressions (extending the CPS transformation of Section 3.1):

$$\mathscr{S}[\![\text{let}\, x = e_1 \ \text{in} \ e_2]\!] K = \mathscr{S}[\![e_1]\!] (\lambda x.\mathscr{S}[\![e_2]\!] K).$$

If we view this equation as the result of circumventing a functional accumulator, we can see that it assumes this accumulator never to be applied. But it is easy to construct

a source term where the accumulator would need to be applied—e.g., the following one.

$$\begin{aligned}\mathscr{S}[\![t_0\ (\mathrm{let}\, x = t_1\ \mathrm{in}\ t_2)]\!]K &= \mathscr{S}[\![(\mathrm{let}\, x = t_1\ \mathrm{in}\ t_2)]\!](\lambda x_1.\mathscr{T}[\![t_0]\!]x_1\, K)\\ &= \mathscr{S}[\![t_1]\!](\lambda x.\mathscr{S}[\![t_2]\!]\ (\lambda x_1.\mathscr{T}[\![t_0]\!]x_1\, K))\\ &= \mathscr{S}[\![t_1]\!](\lambda x.(\lambda x_1.\mathscr{T}[\![t_0]\!]x_1\, K)\,\mathscr{T}[\![t_2]\!])\\ &= (\lambda x.(\lambda x_1.\mathscr{T}[\![t_0]\!]x_1\, K)\,\mathscr{T}[\![t_2]\!])\,\mathscr{T}[\![t_1]\!].\end{aligned}$$

The resulting term is semantically correct, but syntactically it contains an extraneous administrative redex, namely $(\lambda x_1.\mathscr{T}[\![t_0]\!]K)\,\mathscr{T}[\![t_2]\!]$.

In contrast, a higher-order one-pass CPS transformation yields the following more compact term, corresponding to what one might write by hand (with the provision that one usually writes a let expression rather than a β-redex).

$$\mathscr{S}[\![t_0\ (\mathrm{let}\, x = t_1\ \mathrm{in}\ t_2)]\!]k =_\beta (\lambda x.\mathscr{T}[\![t_0]\!]\mathscr{T}[\![t_2]\!]k)\,\mathscr{T}[\![t_1]\!].$$

The CPS transformation of the second edition of *Essentials of Programming Languages* inherits this shortcoming for non-tail let expressions containing computations in their header (i.e. for non-simple let expressions that are not in tail position, to use the terminology of the book).

7.4. Summary and conclusion

We have presented a one-pass CPS transformation that is both first-order and compositional. This CPS transformation makes it possible to reason about CPS-transformed programs by structural induction over source programs. Its correctness proof (i.e. the proof of its simulation theorem) is correspondingly very simple. Elsewhere [28], the second author presents a new and simpler correctness proof of the converse transformation, i.e. the direct-style transformation [7]. Finally, this new CPS transformation has enabled Damian and Danvy to define a one-pass CPS transformation of control-flow information [4,5].

Acknowledgements

Thanks are due to Dan Friedman for a substantial e-mail discussion with the first author about compositionality in the summer of 2000, and to Amr Sabry for a similar discussion at the Third ACM SIGPLAN Workshop on Continuations, in January 2001. This article results from an attempt at unifying our points of view, and has benefited from comments by Daniel Damian, Andrzej Filinski, Mayer Goldberg, Julia Lawall, David Toman, and the anonymous referees. Special thanks to Julia Lawall for a substantial round of proof-reading.

References

[1] A.W. Appel, Compiling with Continuations, Cambridge University Press, New York, 1992.
[2] H. Barendregt, The Lambda Calculus: its Syntax and Semantics, Studies in Logic and the Foundation of Mathematics, Revised Edition, Vol. 103, North-Holland, Amsterdam, 1984.
[3] W.D. Clinger, Proper tail recursion and space efficiency, in: K.D. Cooper (Ed.), Proc. ACM SIGPLAN'98 Conf. on Programming Languages Design and Implementation, Montréal, Canada, ACM Press, Metals Park, June 1998, pp. 174–185.
[4] D. Damian, On static and dynamic control-flow information in program analysis and transformation, Ph.D. Thesis, BRICS Ph.D. School, University of Aarhus, Aarhus, Denmark, July 2001, BRICS DS-01-5.
[5] D. Damian, O. Danvy, A simple CPS transformation of control-flow information, Logic J. IGPL 10 (2002) 501–515.
[6] D. Damian, O. Danvy, Syntactic accidents in program analysis: on the impact of the CPS transformation, J. Funct. Programming, 2002, Extended version available as the Tech. Report BRICS-RS-01-54, to appear.
[7] O. Danvy, Back to direct style, Sci. Comput. Programming 22 (3) (1994) 183–195.
[8] O. Danvy, Formalizing implementation strategies for first-class continuations, in: G. Smolka (Ed.), Proc. 9th European Symp. on Programming, Lecture Notes in Computer Science, Vol. 1782, Springer, Berlin, Germany, March 2000, pp. 88–103.
[9] O. Danvy, B. Dzafic, F. Pfenning. On proving syntactic properties of CPS programs, in: 3rd Internat. Workshop on Higher-Order Operational Techniques in Semantics, Electronic Notes in Theoretical Computer Science, Vol. 26, Paris, France, September 1999, pp. 19–31, also available as the Tech. Report BRICS RS-99-23.
[10] O. Danvy, A. Filinski, Representing control, a study of the CPS transformation, Math. Struct. Comput. Sci. 2 (4) (1992) 361–391.
[11] O. Danvy, L.R. Nielsen, CPS transformation of beta-redexes, in: A. Sabry (Ed.), Proc. 3rd ACM SIGPLAN Workshop on Continuations, Tech. Report 545, Computer Science Department, Indiana University, London, England, January 2001, pp. 35–39, also available as the Tech. Report BRICS RS-00-35.
[12] O. Danvy, L.R. Nielsen, Defunctionalization at work, in: H. Søndergaard (Ed.), Proc. 3rd Internat. Conf. on Principles and Practice of Declarative Programming, Firenze, Italy, ACM Press, Metals Park, September 2001, pp. 162–174, Extended version available as the Tech. Report BRICS RS-01-23.
[13] O. Danvy, L.R. Nielsen, A higher-order colon translation, in: H. Kuchen, K. Ueda (Eds.), Functional and Logic Programming, 5th Internat. Symp., FLOPS 2001, Lecture Notes in Computer Science, Vol. 2024, Tokyo, Japan, Springer, Berlin, March 2001, pp. 78–91, Extended version available as the Tech. Report BRICS RS-00-33.
[14] O. Danvy, L.R. Nielsen, Syntactic theories in practice, in: M. van den Brand, R.M. Verma, (Eds.), Informal Proceedings 2nd Internat. Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4, Firenze, Italy, September 2001, Extended version available as the Tech. Report BRICS RS-02-04.
[15] O. Danvy, F. Pfenning, The occurrence of continuation parameters in CPS terms, Tech. Report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.
[16] B. Dzafic, Formalizing program transformations, Master's Thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.
[17] M. Felleisen, The calculi of λ-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages, Ph.D. Thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.
[18] D.P. Friedman, M. Wand, C.T. Haynes, Essentials of Programming Languages, The MIT Press, McGraw-Hill, Cambridge, New York, 1991.
[19] D.P. Friedman, M. Wand, C.T. Haynes, Essentials of Programming Languages, 2nd Edition, The MIT Press, Cambridge, MA, 2001.
[20] J. Hatcliff, The structure of continuation-passing styles, Ph.D. Thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[21] J. Hatcliff, O. Danvy, A generic account of continuation-passing styles, in: H.J. Boehm (Ed.), Proc. 21st Ann. ACM Symp. on Principles of Programming Languages, Portland, Oregon, ACM Press, New York, January 1994, pp. 458–471.
[22] J.-T. Kim, K. Yi, Interconnecting between CPS terms and non-CPS terms, in: A. Sabry (Ed.), Proc. Third ACM SIGPLAN Workshop on Continuations, Tech. Report 545, Computer Science Department, Indiana University, London, England, January 2001, pp. 7–16.
[23] D. Kranz, R. Kesley, J. Rees, P. Hudak, J. Philbin, N. Adams, Orbit: an optimizing compiler for Scheme, in: Proc. ACM SIGPLAN'86 Symp. on Compiler Construction, Palo Alto, CA, ACM Press, New York, June 1986, pp. 219–233.
[24] A.R. Meyer, M. Wand, Continuation semantics in typed lambda-calculi (summary), in: R. Parikh (Ed.), Logics of Programs—Proc. Lecture Notes in Computer Science, Vol. 193, Brooklyn, New York, Springer, Berlin, June 1985, pp. 219–224.
[25] E. Moggi, Notions of computations and monads, Inform. Comput. 93 (1991) 55–92.
[26] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, ACM Trans. Programming Languages Systems 21 (3) (1999) 527–568.
[27] L.R. Nielsen, A study of defunctionalization and continuation-passing style, Ph.D. Thesis, BRICS Ph.D. School, University of Aarhus, Aarhus, Denmark, July 2001, BRICS DS-01-7.
[28] L.R. Nielsen, A simple correctness proof of the direct-style transformation, Tech. Report BRICS RS-02-02, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002.
[29] I. Ogata, A CPS-transform of constructive classical logic, in: P.S. Thiagarajan, R.H.C. Yap (Eds.), Advances in Computing Science—ASIAN'99, 5th Asian Computing Science Conf., Lecture Notes in Computer Science, Vol. 1742, Phuket, Thailand, Springer, Berlin, December 1999, pp. 266–280.
[30] J. Palsberg, M. Wand, CPS transformation of flow information, J. Funct. Programming, 2002, to appear.
[31] G.D. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoret. Comput. Sci. 1 (1975) 125–159.
[32] J. Polakow, Ordered Linear Logic and Applications, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2001, Tech. Report CMU-CS-01-152.
[33] J. Polakow, F. Pfenning, Natural deduction for intuitionistic non-commutative linear logic, in: J.-Y. Girard (Ed.), Proc. 4th Internat. Conf. on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science, Vol. 1581, L'Aquila, Italy, Springer, Berlin, April 1999, pp. 295–309.
[34] J. Polakow, F. Pfenning, Properties of terms in continuation passing style in an ordered logical framework, in: J. Despeyroux (Ed.), Workshop on logical Frameworks and Meta-Languages (LFM 2000), Santa Barbara, CA, June 2000, www-sop.inria.fr/certilab/LFM00/Proceedings/ .
[35] J. Polakow, K. Yi, Proving syntactic properties of exceptions in an ordered logical framework, in: H. Kuchen, K. Ueda (Eds.), Functional and Logic Programming, 5th Internat. Symp., FLOPS 2001, Lecture Notes in Computer Science, Vol. 2024, Tokyo, Japan, Springer, Berlin, March 2001, pp. 61–77.
[36] J.C. Reynolds, The discoveries of continuations, Lisp Symbol. Comput. 6 (3/4) (1993) 233–247.
[37] J.C. Reynolds, Definitional interpreters for higher-order programming languages, Higher-Order Symbol. Comput. 11 (4) (1998) 363–397, Reprinted from the Proc. 25th ACM National Conference, 1972.
[38] A. Sabry, The formal relationship between direct and continuation passing style optimizing compilers: a synthesis of two paradigms, Ph.D. Thesis, Computer Science Department, Rice University, Houston, TX, August 1994, Tech. Report 94-242.
[39] A. Sabry, M. Felleisen, Reasoning about programs in continuation-passing style, Lisp Symbol. Comput. 6 (3/4) (1993) 289–360.
[40] A. Sabry, P. Wadler, A reflection on call-by-value, ACM Trans. Programming Languages Systems 19 (6) (1997) 916–941.
[41] G.L. Steele Jr. Rabbit: a compiler for Scheme, Master's Thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978, Tech. Report AI-TR-474.
[42] M. Wand, Embedding type structure in semantics, in: M.S. Van Deusen, Z. Galil (Eds.), Proc. 12th Ann. ACM Symp. on Principles of Programming Languages, New Orleans, Louisiana, ACM Press, New York, January 1985, pp. 1–6.

[43] M. Wand, Correctness of procedure representations in higher-order assembly language, in: S. Brookes, M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science, Vol. 598, Pittsburgh, Pennsylvania, Springer, Berlin, March 1991, pp. 294–311; Proc. 7th International Conference.
[44] G. Winskel, The Formal Semantics of Programming Languages, Foundation of computing series, The MIT Press, Cambridge, 1993.
[45] S. Zdancewic, A. Myers, Secure information flow via linear continuations, Higher-Order Symbol. Comput. 15(2/3) (2002) 209–234.