BRICS RS-01-49 Danvy & Nielsen: A First-Order One-Pass CPS Transformation

BRICS

Basic Research in Computer Science

A First-Order One-Pass CPS Transformation

Olivier Danvy
Lasse R. Nielsen

BRICS Report Series **RS-01-49**

ISSN 0909-0878 **December 2001**

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/01/49/`

A First-Order One-Pass CPS Transformation *

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

December 2001

Abstract

We present a new transformation of call-by-value lambda-terms into continuation-passing style (CPS). This transformation operates in one pass and is both compositional and first-order. Because it operates in one pass, it directly yields compact CPS programs that are comparable to what one would write by hand. Because it is compositional, it allows proofs by structural induction. Because it is first-order, reasoning about it does not require the use of a logical relation.

This new CPS transformation connects two separate lines of research. 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.

---

*Extended version of an article to appear in the proceedings of FOSSACS'02, Grenoble, France, April 2002.

†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

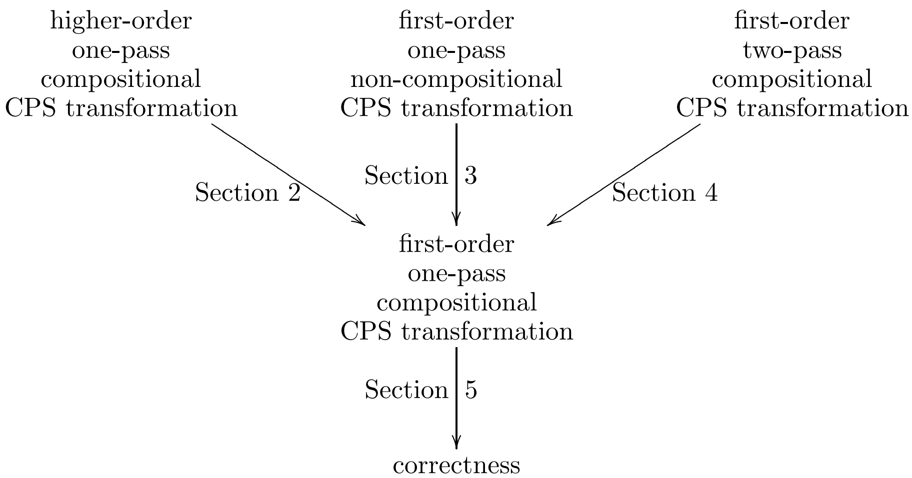
List of Figures

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 [30, 36]. As already reviewed by Reynolds [35], continuations and the CPS transformation share a long history. The CPS transformation was first formalized by Plotkin [30], and first used in practice by Steele, in the first compiler for the Scheme programming language [39]. Unfortunately, its direct implementation as a rewriting system yields extraneous redexes known as *administrative redexes*. These redexes interfere both with proving the correctness of a CPS transformation [30] and with using it in a compiler [22, 39]. At the turn of the 1990's, 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, 41]. The other is non-compositional and first-order, using evaluation contexts [37]. They have both been proven correct and are used in compilers as well as to reason about CPS programs.

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 thus 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 compare 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 [36] and has been used by Moggi to distinguish between values and computations [25].)

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

We distinguish terms up to α-equivalence, i.e., renaming of bound variables.

2 From higher-order to first-order

2.1 A higher-order specification

Figure 1 displays a higher-order, one-pass, compositional CPS transformation.

$$
\begin{array}{rcl}
\mathcal{E} : \mathrm{Expr} & \rightarrow & \mathrm{Ide} \rightarrow \mathrm{Comp} \\
\mathcal{E}[\![t]\!] & = & \overline{\lambda} k.k \mathbin{\underline{@}} \mathcal{T}[\![t]\!] \\
\mathcal{E}[\![s]\!] & = & \overline{\lambda} k.\mathcal{S}[\![s]\!] \mathbin{\overline{@}} k \\
\\
\mathcal{S} : \mathrm{Comp} & \rightarrow & \mathrm{Ide} \rightarrow \mathrm{Comp} \\
\mathcal{S}[\![e_0\ e_1]\!] & = & \overline{\lambda} k.\mathcal{E}'[\![e_0]\!] \mathbin{\overline{@}} (\overline{\lambda} x_0.\mathcal{E}'[\![e_1]\!] \mathbin{\overline{@}} (\overline{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} k)) \\
\\
\mathcal{T} : \mathrm{Val} & \rightarrow & \mathrm{Val} \\
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![\lambda x.e]\!] & = & \underline{\lambda} x.\underline{\lambda} k.\mathcal{E}[\![e]\!] \mathbin{\overline{@}} k \\
\\
\mathcal{E}' : \mathrm{Expr} & \rightarrow & (\mathrm{Val} \rightarrow \mathrm{Comp}) \rightarrow \mathrm{Comp} \\
\mathcal{E}'[\![t]\!] & = & \overline{\lambda} \kappa.\kappa \mathbin{\overline{@}} \mathcal{T}[\![t]\!] \\
\mathcal{E}'[\![s]\!] & = & \overline{\lambda} \kappa.\mathcal{S}'[\![s]\!] \mathbin{\overline{@}} \kappa \\
\\
\mathcal{S}' : \mathrm{Comp} & \rightarrow & (\mathrm{Val} \rightarrow \mathrm{Comp}) \rightarrow \mathrm{Comp} \\
\mathcal{S}'[\![e_0\ e_1]\!] & = & \overline{\lambda} \kappa.\mathcal{E}'[\![e_0]\!] \mathbin{\overline{@}} (\overline{\lambda} x_0.\mathcal{E}'[\![e_1]\!] \mathbin{\overline{@}} (\overline{\lambda} x_1.x_0 \mathbin{\underline{@}} x_1 \mathbin{\underline{@}} (\underline{\lambda} x_2.\kappa \mathbin{\overline{@}} x_2)))
\end{array}
$$

Figure 1: Higher-order one-pass CPS transformation

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

In Figure 1, transformation-time abstractions ($\overline{\lambda}$) and applications (infix $\overline{@}$) are overlined. Underlined abstractions ($\underline{\lambda}$) and applications (infix $\underline{@}$) are hygienic syntax constructors, i.e., they generate fresh variables.

An expression e is CPS-transformed into the result of $\underline{\lambda} k.\mathcal{E}[\![e]\!] \,\overline{@}\, k$.

2.2 Circumventing the higher-order functions

Let us analyze the function spaces in Figure 1. All the calls to $\mathcal{E}$, $\mathcal{S}$, $\mathcal{E}'$, and $\mathcal{S}'$ are fully applied and thus these functions could as well be uncurried. The resulting CPS transformation is only higher order because of the function space $\mathrm{Val} \rightarrow \mathrm{Comp}$ used in $\mathcal{E}'$ and $\mathcal{S}'$. Let us try to circumvent this function space.

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

$$
\begin{tikzcd}
\mathcal{E} \arrow[r] \arrow[dr, bend left] & \mathcal{S} \arrow[r] & \mathcal{E}' \arrow[r, bend left] \arrow[dl] & \mathcal{S}' \arrow[l, bend left] \\
& \mathcal{T} \arrow[ul, bend left] & &
\end{tikzcd}
$$

Therefore, if we could prevent $\mathcal{S}$ from calling $\mathcal{E}'$, both $\mathcal{E}'$ and $\mathcal{S}'$ would become dead code, and only $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{T}$ would remain. We would then obtain a first-order one-pass CPS transformation.

Let us unfold the definition of $\mathcal{S}$ and reason by inversion. The four following cases occur. (We only detail the β-reductions in the first case.)

$$
\begin{array}{lll}
\mathcal{S}[\![t_0\ t_1]\!] \,\overline{@}\, k & =_{\mathrm{def}} & \mathcal{E}'[\![t_0]\!] \,\overline{@}\, (\overline{\lambda} x_0.\mathcal{E}'[\![t_1]\!] \,\overline{@}\, (\overline{\lambda} x_1.x_0 \,\underline{@}\, x_1 \,\underline{@}\, k)) \\
& =_{\mathrm{def}} & (\overline{\lambda} x_0.\mathcal{E}'[\![t_1]\!] \,\overline{@}\, (\overline{\lambda} x_1.x_0 \,\underline{@}\, x_1 \,\underline{@}\, k)) \,\overline{@}\, \mathcal{T}[\![t_0]\!] \\
& \rightarrow_{\beta} & \mathcal{E}'[\![t_1]\!] \,\overline{@}\, (\overline{\lambda} x_1.\mathcal{T}[\![t_0]\!] \,\underline{@}\, x_1 \,\underline{@}\, k) \\
& =_{\mathrm{def}} & (\overline{\lambda} x_1.\mathcal{T}[\![t_0]\!] \,\underline{@}\, x_1 \,\underline{@}\, k) \,\overline{@}\, \mathcal{T}[\![t_1]\!] \\
& \rightarrow_{\beta} & \mathcal{T}[\![t_0]\!] \,\underline{@}\, \mathcal{T}[\![t_1]\!] \,\underline{@}\, k \\
\\
\mathcal{S}[\![t_0\ s_1]\!] \,\overline{@}\, k & =_{\beta} & \mathcal{S}'[\![s_1]\!] \,\overline{@}\, (\overline{\lambda} x_1.\mathcal{T}[\![t_0]\!] \,\underline{@}\, x_1 \,\underline{@}\, k) \\
\\
\mathcal{S}[\![s_0\ t_1]\!] \,\overline{@}\, k & =_{\beta} & \mathcal{S}'[\![s_0]\!] \,\overline{@}\, (\overline{\lambda} x_0.x_0 \,\underline{@}\, \mathcal{T}[\![t_1]\!] \,\underline{@}\, k) \\
\\
\mathcal{S}[\![s_0\ s_1]\!] \,\overline{@}\, k & =_{\beta} & \mathcal{S}'[\![s_0]\!] \,\overline{@}\, (\overline{\lambda} x_0.\mathcal{S}'[\![s_1]\!] \,\overline{@}\, (\overline{\lambda} x_1.x_0 \,\underline{@}\, x_1 \,\underline{@}\, k))
\end{array}
$$

This analysis makes explicit all of the functions κ that $\mathcal{S}$ passes to $\mathcal{S}'$. By definition of $\mathcal{S}'$, we also know *where* these functions are applied: in the two-level eta-redex $\underline{\lambda} x_2.\kappa \,\overline{@}\, x_2$. We can take advantage of this knowledge by invoking

$\mathcal{S}$ rather than $\mathcal{S}'$, extend its domain to Comp $\rightarrow$ Expr $\rightarrow$ Comp, and pass it the result of eta-expanding κ. The result reads as follows.

$$
\begin{array}{rcl}
\mathcal{S}[\![t_0\ t_1]\!]\,\overline{@}\,k & \equiv & \mathcal{T}[\![t_0]\!]\,\underline{@}\,\mathcal{T}[\![t_1]\!]\,\underline{@}\,k \\
\mathcal{S}[\![t_0\ s_1]\!]\,\overline{@}\,k & \equiv & \mathcal{S}[\![s_1]\!]\,\overline{@}\,(\underline{\lambda} x_1.\mathcal{T}[\![t_0]\!]\,\underline{@}\,x_1\,\underline{@}\,k) \\
\mathcal{S}[\![s_0\ t_1]\!]\,\overline{@}\,k & \equiv & \mathcal{S}[\![s_0]\!]\,\overline{@}\,(\underline{\lambda} x_0.x_0\,\underline{@}\,\mathcal{T}[\![t_1]\!]\,\underline{@}\,k) \\
\mathcal{S}[\![s_0\ s_1]\!]\,\overline{@}\,k & \equiv & \mathcal{S}[\![s_0]\!]\,\overline{@}\,(\underline{\lambda} x_0.\mathcal{S}[\![s_1]\!]\,\overline{@}\,(\underline{\lambda} x_1.x_0\,\underline{@}\,x_1\,\underline{@}\,k))
\end{array}
$$

In this derived transformation, $\mathcal{E}'$ and $\mathcal{S}'$ 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.

$$
\begin{tikzcd}
\mathcal{E} \arrow[r] \arrow[dr, bend left] & \mathcal{S} \arrow[loop right] \arrow[d] \\
& \mathcal{T} \arrow[ul, bend left]
\end{tikzcd}
$$

The resulting CPS transformation is displayed in Figure 2. Since it is first-order, there are no overlined abstractions and applications, and therefore we omit all underlines as well as the infix @. An expression e is CPS-transformed into the result of $\lambda k.\mathcal{E}[\![e]\!]\,k$.

$$
\begin{array}{rcl}
\mathcal{E} : \text{Expr} \times \text{Ide} & \rightarrow & \text{Comp} \\
\mathcal{E}[\![t]\!]\,k & = & k\ \mathcal{T}[\![t]\!] \\
\mathcal{E}[\![s]\!]\,k & = & \mathcal{S}[\![s]\!]\,k \\
\\
\mathcal{S} : \text{Comp} \times \text{Expr} & \rightarrow & \text{Comp} \\
\mathcal{S}[\![t_0\ t_1]\!]\,K & = & \mathcal{T}[\![t_0]\!]\ \mathcal{T}[\![t_1]\!]\ K \\
\mathcal{S}[\![t_0\ s_1]\!]\,K & = & \mathcal{S}[\![s_1]\!]\ (\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K) \\
\mathcal{S}[\![s_0\ t_1]\!]\,K & = & \mathcal{S}[\![s_0]\!]\ (\lambda x_0.x_0\ \mathcal{T}[\![t_1]\!]\ K) \\
\mathcal{S}[\![s_0\ s_1]\!]\,K & = & \mathcal{S}[\![s_0]\!]\ (\lambda x_0.\mathcal{S}[\![s_1]\!]\ (\lambda x_1.x_0\ x_1\ K)) \\
\\
\mathcal{T} : \text{Val} & \rightarrow & \text{Val} \\
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![\lambda x.e]\!] & = & \lambda x.\lambda k.\mathcal{E}[\![e]\!]\,k
\end{array}
$$

Figure 2: First-order one-pass CPS transformation

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 [42, page 60]. One could say, however, that it is not purely defined by recursive descent, since $\mathcal{S}$ is defined by cases on immediate sub-expressions, 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 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

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, 37]. Sabry and Wadler then simplified this CPS transformation [38, Figure 18], e.g., 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 Figure 2, Sabry and Wadler's CPS transformation reads as follows. An expression e is CPS-transformed into $\lambda k.\mathcal{E}[\![e]\!]$, where:

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

For each serious expression s with a serious immediate sub-expression s', $\mathcal{S}$ recursively traverses s' with a new continuation. In this new continuation, s' is replaced by a fresh variable (i.e., a trivial immediate sub-expression) in s. The result, now with one less serious immediate sub-expression, is transformed recursively. The idea was the same in Sabry and Felleisen's context-based CPS transformation [37, 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

$\mathcal{S}$ 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 proving termination therefore requires a size argument. 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 $\mathcal{S}$ in $\mathcal{S}[\![t_0\ x_1]\!]\,K$: The result is $\mathcal{T}[\![t_0]\!]\ \mathcal{T}[\![x_1]\!]\ K$, which is equivalent to $\mathcal{T}[\![t_0]\!]\ x_1\ K$.

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

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

The resulting unfolded transformation is compositional. It also coincides with the definition of $\mathcal{S}$ in Figure 2 and thus connects the two separate lines of research.

4 From two passes to one pass

4.1 A two-pass specification

Plotkin's CPS transformation [30] can be phrased as follows.

$$\begin{array}{rcl}
\mathcal{C}[\![t]\!] & = & \lambda k.k\ \Phi(t) \\
\mathcal{C}[\![e_0\ e_1]\!] & = & \lambda k.\mathcal{C}[\![e_0]\!]\ (\lambda x_0.\mathcal{C}[\![e_1]\!]\ (\lambda x_1.x_0\ x_1\ k)) \\[1ex]
\Phi(x) & = & x \\
\Phi(\lambda x.e) & = & \lambda x.\mathcal{C}[\![e]\!]
\end{array}$$

Directly implementing it yields CPS terms containing a mass of administrative redexes that need to be contracted in a second pass [39].

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.

$$
\begin{tikzcd}
e \arrow[r, "\text{reduction}"] \arrow[d, "\text{CPS transformation}"'] & e' \arrow[d, "\text{CPS transformation}"] \\
\mathcal{C}[\![e]\!]K \arrow[d, "\begin{array}{c}\text{administrative}\\\text{reductions}\end{array}"'] & \mathcal{C}[\![e']\!]K \arrow[d, "\begin{array}{c}\text{administrative}\\\text{reductions}\end{array}"] \\
e : K \arrow[r, "\text{reduction}"] & e' : K
\end{tikzcd}
$$

The colon translation is itself a CPS transformation. It transforms a source expression 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 expression applied to the continuation. In other words, if we write the colon translation of the expression e and the continuation K as $e : K$, then the following holds: $\mathcal{C}[\![e]\!]\ K \overset{*}{\rightarrow} e : K$.

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 $\lambda k.(\lambda k.k\ \Phi(v))\ (\lambda x_0.\mathcal{C}[\![e_1]\!]\ (\lambda x_1.x_0\ x_1\ k))$. Applying this CPS term to a continuation enables the following administrative reductions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\lambda k.(\lambda k.k\ \Phi(t_0))\ (\lambda x_0.\mathcal{C}[\![e_1]\!]\ (\lambda x_1.x_0\ x_1\ k)))\ K} \\
\quad\rightarrow_\beta & (\lambda k.k\ \Phi(t_0))\ (\lambda x_0.\mathcal{C}[\![e_1]\!]\ (\lambda x_1.x_0\ x_1\ K)) \\
\quad\rightarrow_\beta & (\lambda x_0.\mathcal{C}[\![e_1]\!]\ (\lambda x_1.x_0\ x_1\ K))\ \Phi(t_0) \\
\quad\rightarrow_\beta & \mathcal{C}[\![e_1]\!]\ \lambda x_1.\Phi(t_0)\ x_1\ K
\end{array}
$$

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

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

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, further administrative redexes can be contracted by using the colon translation in place of all occurrences of $\mathcal{C}$.

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 expression 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.

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

With a change of notation, this extended colon translation coincides with the first-order one-pass CPS transformation from Figure 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 Correctness of the first-order one-pass CPS transformation

We prove the correctness of the transformation of Figure 2 in the traditional way established by Plotkin [30]. To this end, we first define a reduction relation on programs and then we prove a simulation theorem.

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.

$$\mathrm{E} \quad ::= \quad [\,] \mid \mathrm{E}\ e \mid t\ \mathrm{E}$$

A context is an expression with a hole. We plug the hole of a context E with an expression e (noted $\mathrm{E}[e]$) as follows.

$$
\begin{array}{rcl}
[\,][e] & = & e \\
(\mathrm{E}\ e')[e] & = & (\mathrm{E}[e])\ e' \\
(t\ \mathrm{E})[e] & = & t\ (\mathrm{E}[e])
\end{array}
$$

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

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

and this decomposition is unique.

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

$$\mathrm{E}[(\lambda x.e)\ t] \rightarrow \mathrm{E}[e[t/x]]$$

where $e[t/x]$ is the usual capture-avoiding substitution of t for free occurrences of x in e. We call an expression on the form $(\lambda x.e)\ t$ a redex.

We say that e is reducible if there exists an e' such that $e \rightarrow e'$. So, for example, values are not reducible. We write $\xrightarrow{+}$, $\xrightarrow{*}$, and $\xrightarrow{n}$ for the transitive closure, the reflexive and transitive closure, and the n-times composition of the relation $\rightarrow$.

Some computations are not reducible. They are said to "stick". The set of stuck terms is exactly those on the form $\mathrm{E}[x\ t]$, i.e., the application of a *variable* to a value in an evaluation context. Since the decomposition is unique, such an expression cannot be reducible.

5.2 Simulation

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 an expression, t a value, x a variable, and K another value then*

$$(\mathcal{E}[\![e]\!]\,K)[\mathcal{T}[\![t]\!]/x] \quad = \quad \mathcal{E}[\![e[t/x]]\!]\,(K[\mathcal{T}[\![t]\!]/x])$$

If e is an expression, k is a variable, and K an expression then

$$(\mathcal{E}[\![e]\!]\,k)[K/k] \quad = \quad \mathcal{E}[\![e]\!]\,K$$

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

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

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

$$e \rightarrow e' \Longrightarrow \mathcal{E}[\![e]\!]\,K \xrightarrow{+} \mathcal{E}[\![e']\!]\,K$$

Proof: If $e \rightarrow e'$ then there exists a context E, a redex $t_0\ t_1$, and an expression e'' such that $e = \mathrm{E}[t_0\ t_1]$ and $e' = \mathrm{E}[e'']$. The proof, which we omit, is by induction on the context E. QED

Lemma 2 accounts for all reducible expressions. The following lemma handles the expressions that stick.

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

Proof: Since all stuck expressions are of the form E$[x\ t]$, the proof is by induction on E. QED

Theorem 1 (Simulation) *If e is an expression and v is a CPS value then*

$$(\exists t.e \xrightarrow{*} t \wedge \mathcal{T}[\![t]\!] = v) \iff \mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v$$

Proof:

1. $(\exists t.e \xrightarrow{*} t \wedge \mathcal{T}[\![t]\!] = v) \Longrightarrow \mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v$

   Let e, v, and t be given with $v = \mathcal{T}[\![t]\!]$. From repeated use of Lemma 2, it follows that $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} \mathcal{E}[\![t]\!]\,\lambda x.x$, and $\mathcal{E}[\![t]\!]\,\lambda x.x = (\lambda x.x)\ \mathcal{T}[\![t]\!] \rightarrow \mathcal{T}[\![t]\!] = v$.

2. $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v \Longrightarrow (\exists t.e \xrightarrow{*} t \wedge \mathcal{T}[\![t]\!] = v)$

   Let e and v be given. Then this implication is proved by contraposition, i.e., assume that $\exists t.e \xrightarrow{*} t \wedge \mathcal{T}[\![t]\!] = v$ fails to hold. Either there is no t such that $e \xrightarrow{*} t$, or there is one, but $\mathcal{T}[\![t]\!] \neq v$.

   The expression e does not reduce to a value t in two cases: when e diverges, i.e., has an infinite derivation, and when it reduces to a stuck term. In either case $\mathcal{E}[\![e]\!]\,\lambda x.x$ has the same behavior.

   - e diverges $\Longrightarrow \mathcal{E}[\![e]\!]\,\lambda x.x$ diverges

     If an expression 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 expression). That is, for all numbers n there exists an expression e_n such that $e \xrightarrow{n} e_n$.

     Now, let n be a natural number. We consider the sequence $e \rightarrow e_1 \rightarrow \cdots \rightarrow e_n$, which exists since e diverges. Then, from Lemma 2 we know that there is another reduction sequence $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{+} \mathcal{E}[\![e_1]\!]\,\lambda x.x \xrightarrow{+} \cdots \xrightarrow{+} \mathcal{E}[\![e_n]\!]\,\lambda x.x$ of length at least n. Therefore $\mathcal{E}[\![e]\!]\,\lambda x.x$ has reduction sequences of arbitrary length and thus it diverges as well.

   - $e \xrightarrow{*} e'$ and e' sticks $\Longrightarrow \mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} e''$ and e'' sticks

     From repeated use of Lemma 2 we know that $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} \mathcal{E}[\![e']\!]\,\lambda x.x$, and from Lemma 3 we know that $\mathcal{E}[\![e']\!]\,\lambda x.x$ sticks.

- Let t be given such that $e \xrightarrow{*} t \wedge \mathcal{T}[\![t]\!] \neq v$. Then it cannot be the case that $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} v$. This follows from the implication in the other direction, since then $\mathcal{E}[\![e]\!]\,\lambda x.x \xrightarrow{*} \mathcal{T}[\![t]\!] \neq v$.

Together these cases account for all possible reductions of an expression, which suffices to prove the simulation theorem. QED

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. As our running example, we prove that the CPS transformation preserves types. (The CPS transformation of types exists [24, 40] and has a logical content [20, 26].) 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 [31, 32, 33].) 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 preserves types: To prove the well-typedness of a CPS-transformed term, we proceed by structural induction on the typing derivation of the source term (or by structural induction on the source expression), 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 [37, 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 [31, 34].

The CPS transformation preserves types: To prove the well-typedness of a CPS-transformed term, 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 [30] 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, 29], 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 preserves types: To prove the well-typedness of a CPS-transformed term, we proceed by structural induction on the typing derivation of the source term.

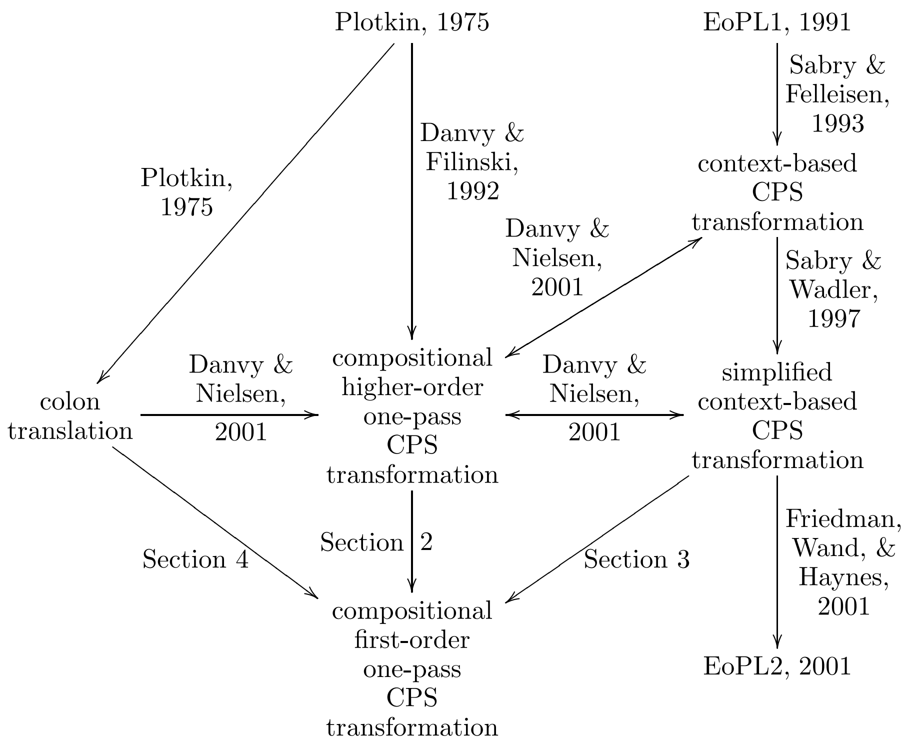
6.4 Non-compositional CPS transformations

Sabry and Felleisen's proofs are by induction on the size of the source program [37, Appendix A, page 337]. Proving type preservation would require a substitution lemma.

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 Figure 1).

$$\begin{array}{rcl} \mathcal{S}[\![\text{let } x = e_1 \text{ in } e_2]\!] & = & \overline{\lambda} k.\mathcal{E}[\![e_1]\!] \,\overline{@}\, (\underline{\lambda} x.\mathcal{E}[\![e_2]\!] \,\overline{@}\, k) \\ \mathcal{S}'[\![\text{let } x = e_1 \text{ in } e_2]\!] & = & \overline{\lambda} \kappa.\mathcal{E}'[\![e_1]\!] \,\overline{@}\, (\underline{\lambda} x.\mathcal{E}'[\![e_2]\!] \,\overline{@}\, \kappa) \end{array}$$

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. Therefore no 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 [38] also handles let expressions (extending the CPS transformation of Section 3.1):

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

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.

$$\begin{array}{rcl} \mathcal{S}[\![t_0 \ (\text{let } x = t_1 \text{ in } t_2)]\!]\, K & = & \mathcal{S}[\![\text{let } x = t_1 \text{ in } t_2]\!]\, (\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K) \\ & = & \mathcal{S}[\![t_1]\!]\, (\lambda x.\mathcal{S}[\![t_2]\!]\, (\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K)) \\ & = & \mathcal{S}[\![t_1]\!]\, (\lambda x.(\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K)\ \mathcal{T}[\![t_2]\!]) \\ & = & (\lambda x.(\lambda x_1.\mathcal{T}[\![t_0]\!]\ x_1\ K)\ \mathcal{T}[\![t_2]\!])\ \mathcal{T}[\![t_1]\!] \end{array}$$

The resulting term is semantically correct, but syntactically it contains an extraneous administrative redex.

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).

$$\mathcal{S}[\![t_0 \ (\text{let } x = t_1 \text{ in } t_2)]\!]\, k \quad \equiv \quad (\lambda x.\mathcal{T}[\![t_0]\!]\ \mathcal{T}[\![t_2]\!]\ k)\ \mathcal{T}[\![t_1]\!]$$

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. The second author's PhD thesis [27, 28] also contains 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].

Acknowledgments: 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.

A An example of continuation-passing program

The following ML functions compute the map functional. One is in direct style, and the other one in CPS.

```
(*  map : ('a -> 'b) * 'a list -> 'b list *)
fun map (f, nil)
    = nil
  | map (f, x :: xs)
    = (f x) :: (map (f, xs))

(*  map_c : ('a * ('b -> 'c) -> 'c) * 'a list * ('b list -> 'c) -> 'c *)
fun map_c (f_c, nil, k)
    = k nil
  | map_c (f_c, x :: xs, k)
    = f_c (x, fn v => map_c (f_c, xs, fn vs => k (v :: vs)))
```

The direct-style function `map` takes a direct-style function and a list as arguments, and yields another list as result.

The continuation-passing function `map_c` takes a continuation-passing function, a list, and a continuation as arguments. It yields a result of type `'c`, which is also the type of the final result of any CPS program that uses `map_c`. Matching the result type `'b list` of `map`, the continuation of `map_c` has type `'b list -> 'c`. Matching the argument type `'a -> 'b` of `map`, the first argument of `map_c` is a continuation-passing function of type `'a * ('b -> 'c) -> 'c`.

In the base case, `map` returns `nil` whereas `map_c` sends `nil` to the continuation. For a non-empty list, `map` constructs a list with the result of its first argument on the head of the list and with the result of a recursive call on the rest of the list. In contrast, `map_c` calls its first argument on the head of the list with a new continuation that, when sent a result, recursively calls `map_c` on the rest of the list with a new continuation that, when sent a list of results, constructs a list and sends it to the continuation. In `map_c`, all calls are tail calls.

References

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

[2] 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.

[3] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[4] Daniel Damian. *On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-5.

[5] Daniel Damian and Olivier Danvy. A simple CPS transformation of control-flow information. Technical Report BRICS RS-01-55, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2001.

[6] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. *Journal of Functional Programming*, 2002. To appear. Extended version available as the technical report BRICS-RS-01-54.

[7] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[8] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[9] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. 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. Also available as the technical report BRICS RS-99-23.

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

[11] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, Technical report 545, Computer Science Department, Indiana University, pages 35–39, London, England, January 2001. Also available as the technical report BRICS RS-00-35.

[12] 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.

[13] Olivier Danvy and Lasse R. Nielsen. A higher-order colon translation. In Kuchen and Ueda [23], pages 78–91. Extended version available as the technical report BRICS RS-00-33.

[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 Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.

[16] Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.

[17] 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.

[18] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press and McGraw-Hill, 1991.

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

[20] 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.

[21] 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.

[22] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM Press.

[23] Herbert Kuchen and Kazunori Ueda, editors. *Functional and Logic Programming, 5th International Symposium, FLOPS 2001*, number 2024 in Lecture Notes in Computer Science, Tokyo, Japan, March 2001. Springer-Verlag.

[24] 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.

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

[26] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

[27] 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.

[28] Lasse R. Nielsen. A simple correctness proof of the direct-style transformation. Technical Report BRICS RS-02-02, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 2002.

[29] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. Unpublished manuscript, available at `http://www.cs.purdue.edu/~palsberg/publications.html`, June 2001.

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

[31] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2001. Technical Report CMU-CS-01-152.

[32] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, number 1581 in Lecture Notes in Computer Science, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag.

[33] Jeff Polakow and Frank Pfenning. Properties of terms in continuation passing style in an ordered logical framework. In Joëlle Despeyroux, editor, *Workshop on Logical Frameworks and Meta-Languages (LFM 2000)*, Santa Barbara, California, June 2000. `http://www-sop.inria.fr/certilab/LFM00/Proceedings/`.

[34] Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. In Kuchen and Ueda [23], pages 61–77.

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

[36] 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).

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

[38] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997.

[39] 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.

[40] Mitchell Wand. Embedding type structure in semantics. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 1–6, New Orleans, Louisiana, January 1985. ACM Press.

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

[42] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

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.**

RS-01-45 **Ivan B. Damgård and Gudmund Skovbjerg Frandsen.** ***An Extended Quadratic Frobenius Primality Test with Average Case Error Estimates*****. November 2001. 43 pp.**

RS-01-44 **M. Oliver Möller, Harald Rueß, and Maria Sorea.** ***Predicate Abstraction for Dense Real-Time Systems*****. November 2001. 27 pp.**

RS-01-43 **Ivan B. Damgård and Jesper Buus Nielsen.** ***From Known-Plaintext Security to Chosen-Plaintext Security*****. November 2001. 18 pp.**

RS-01-42 **Zoltán Ésik and Werner Kuich.** ***Rationally Additive Semirings*****. November 2001. 11 pp.**

RS-01-41 **Ivan B. Damgård and Jesper Buus Nielsen.** ***Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor*****. October 2001. 43 pp.**

RS-01-40 **Daniel Damian and Olivier Danvy.** ***CPS Transformation of Flow Information, Part II: Administrative Reductions*****. October 2001. 9 pp.**

RS-01-39 **Olivier Danvy and Mayer Goldberg.** ***There and Back Again*****. October 2001. 14 pp.**

RS-01-38 **Zoltán Ésik.** ***Free De Morgan Bisemigroups and Bisemilattices*****. October 2001. 13 pp.**

RS-01-37 **Ronald Cramer and Victor Shoup.** ***Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption*****. October 2001. 34 pp.**