

Basic Research in Computer Science

BRICS RS-96-20

Danvy & Lawall: Back to Direct Style II: First-Class Continuations

Back to Direct Style II: First-Class Continuations

Olivier Danvy
Julia L. Lawall

BRICS Report Series

RS-96-20

ISSN 0909-0878

June 1996

**Copyright © 1996, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent publications in the BRICS
Report Series. 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 WWW and
anonymous FTP:**

`http://www.brics.dk/
ftp ftp.brics.dk (cd pub/BRICS)`

Back to Direct Style II: First-Class Continuations ^{*}

Olivier Danvy BRICS [†] Computer Science Department Aarhus University [‡] (danvy@brics.dk)	Julia L. Lawall IRISA University of Rennes [§] (Julia.Lawall@irisa.fr)
---	---

June 1996

Abstract

The direct-style transformation aims at mapping continuation-passing programs back to direct style, be they originally written in continuation-passing style or the result of the continuation-passing-style transformation. In this paper, we continue to investigate the direct-style transformation by extending it to programs with first-class continuations.

First-class continuations break the stack-like discipline of continuations in that they are sent results out of turn. We detect them syntactically through an analysis of continuation-passing terms. We conservatively extend the direct-style transformation towards call-by-value functional terms (the pure λ -calculus) by translating the declaration of a first-class continuation using the control operator `call/cc`, and by translating an occurrence of a first-class continuation using the control operator `throw`. We prove that our extension and the corresponding extended continuation-passing-style transformation are inverses.

^{*}A preliminary version of this paper appeared in the proceedings of the 1992 ACM Conference on Lisp and Functional Programming, William Clinger (editor), LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[†]Basic Research in Computer Science,

Centre of the Danish National Research Foundation.

[‡]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.

[§]Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France. This work was partially supported by ONR under grant N00014-93-1-1015 and by a NSF International Post-Doctoral Research Fellowship.

Both the direct-style (DS) and continuation-passing-style (CPS) transformations can be generalized to a richer language. These transformations have a place in the programmer's toolbox, and enlarge the class of programs that can be manipulated on a semantic basis. We illustrate both with three applications: the conversion between CPS and DS of an interpreter hand-written in CPS; the specialization of a coroutine program, where coroutines are implemented using `call/cc`; and the normalization of programs extracted from classical proofs. The second example achieves a first: a static coroutine is executed statically and its computational content is inlined in the residual program.

Keywords: continuation-passing style transformation, direct-style transformation, control operators, λ -calculus, partial evaluation, Schism, Scheme.

List of Figures

1	Syntax of pure DS terms	7
2	Call-by-value CPS transformation on pure DS terms	8
3	Syntax of CPS terms	8
4	Occurrence conditions over continuations in pure CPS terms	9
5	Call-by-value DS transformation on pure CPS terms	9
6	Occurrence conditions over (possibly first-class) continuations	11
7	Detection of non-first-class continuations	11
8	Free continuation identifiers in a CPS term	11
9	Syntax of DS terms, including call/cc and throw	12
10	Call-by-value DS transformation, including call/cc and throw	12
11	Call-by-value CPS transformation, including call/cc and throw	13
12	Syntax of pure intermediate-style terms	15
13	From direct style to intermediate style	15
14	From intermediate style to continuation-passing style (continuation introduction)	16
15	From continuation-passing style to intermediate style (continuation elimination)	16
16	From intermediate style to direct style (let unfolding)	17
17	Syntax of intermediate-style terms, including call/cc and throw	19
18	From direct style to intermediate style, including call/cc and throw	19
19	From intermediate style to continuation-passing style, including call/cc and throw	20
20	From continuation-passing style to intermediate style, including call/cc and throw	20
21	From intermediate style to direct style, including call/cc and throw	21
22	Control structures for the Samefringe program	26
23	Data structures for the Samefringe program	27
24	Specialized version of the Samefringe program	27
25	Haynes, Friedman, and Wand's CPS interpreter for Scheme 84 (part I)	31
26	Haynes, Friedman, and Wand's CPS interpreter for Scheme 84 (part II)	32
27	Direct-style counterpart of Haynes, Friedman, and Wand's interpreter for Scheme 84	33

1 Introduction

Functional-programming folklore has it that control operators such as call-with-current-continuation (abbreviated *call/cc*) are unnecessary because their effect can be simulated by continuation-passing style (CPS). On the other hand, CPS forces one to write programs in an extraordinarily contrived way. Fortunately, the CPS transformation automatically maps programs (with or without control operators) into purely functional programs.

In “Back to Direct Style” [7], Danvy has shown how to reverse this process, mapping CPS terms back to pure direct-style (DS) terms. The transformation relies on the restricted language of CPS terms that arises from the call-by-value, left-to-right CPS transformation of pure λ -terms. In this paper, we examine the effect of relaxing some of these restrictions. We find that a natural extension of CPS terms to first-class continuations corresponds to DS terms that use the control operator *call/cc*.

First, we describe CPS, first-class continuations, and *call/cc*. We then outline the rest of the paper.

1.1 Continuation-passing style

In an expression language such as the λ -calculus, every computation occurs in a context. Creating new contexts is a natural computational step in such languages. For example, in the term $f x y$, where f denotes a (curried) function and application associates to the left, f is first applied to x , and the result is applied to y . The application of f to x occurs in a new context, often written $[\cdot] y$.

Some terms, however, create no new contexts. For example, in the λ -abstraction $\lambda x.f x y$, the call to the result of $f x$ occurs in the same context as the call to the λ -abstraction, and thus it does not create any new context. The call to f , however, does.

Contexts can be encoded as λ -abstractions. So for example, the context $[\cdot] y$ is represented as the λ -abstraction $\lambda v.v y$. Composing these contexts forms a *continuation*. In the class of *continuation-passing* terms, a function is passed a continuation along with the function’s argument. These terms create no new contexts.

Terms that create new contexts are usually said to be in *direct style* (DS), to contrast them with continuation-passing terms, and by analogy with denotational semantics [41]. Any DS term can be automatically transformed into a CPS term, using the *CPS transformation* [8, 15, 27, 33, 40].

The result of applying the CPS transformation to the λ -term above is $\lambda x.\lambda k.f_c x \lambda v.v y k$, where f_c is the CPS counterpart of f . This CPS transformation is reversible: the *DS transformation* [7] maps this CPS term back to direct style.

To summarize, in CPS, every function is passed a continuation as an additional argument. Each function produces an intermediate result and sends it to its continuation. This continuation describes how to use this intermediate result in the rest of the computation. Consequently, no function call creates a new context.

1.2 First-class continuations

The CPS language generated by the CPS transformation is quite restricted. For example, corresponding to the fact that every DS expression occurs in one context (there is only one current context), every CPS expression accesses only one continuation (there is only one current continuation). In the following, we relax this restriction on CPS terms.

We relax CPS terms by allowing all expressions to use any continuation that is lexically visible — instead of only the current one, which happens to be the one that is lexically closest.

For example, let us consider lambda terms of the following form:

$$\lambda k.f_c (\lambda i.\lambda k_1._ i) \lambda v.g_c v k$$

Here the CPS function f_c is applied to a CPS function argument and a continuation sending the intermediate result to g_c . The argument to f_c is a function that sends its argument i to an as-yet uninstantiated continuation.

If we replace $_$ by k_1 , we obtain

$$\lambda k.f_c (\lambda i.\lambda k_1.k_1 i) \lambda v.g_c v k$$

in which the argument of f_c simply sends its argument to its current continuation. The DS transformation maps this term into

$$g (f \lambda i.i)$$

where f and g are the DS counterparts of f_c and g_c , respectively.

On the other hand, if we replace $_$ by k , we obtain

$$\lambda k.f_c (\lambda i.\lambda k_1.k i) \lambda v.g_c v k$$

in which the argument of f_c sends its intermediate result not to its continuation, but to the continuation of the entire term. We refer to the continuation k , which is used out of turn, as a “first-class continuation”.

In CPS terms with first-class continuations, every function takes a continuation as an argument, and sends its result to some continuation. When this continuation is not the current one, this evaluation step does not correspond to a standard evaluation step in direct style. Such CPS terms cannot be mapped back to a pure DS functional term.

1.3 Call/cc

CPS makes the continuation of each function call explicit as a parameter of the function. The DS control operator `call/cc` provides access to the current continuation at any point while allowing the continuation to remain generally implicit [3]. Variants of `call/cc` are provided by a variety of widely used programming languages that include Lisp, C, Scheme, and Standard ML of New Jersey. In this paper we follow the strategy of Lisp, C, and Standard ML, and include a *throw* operator to apply a continuation that was accessed with `call/cc`.

`Call/cc` and `throw` can describe the behavior of the CPS expression in the previous section:

$$\lambda k.f_c (\lambda i.\lambda k_1.k i) \lambda v.g_c v k$$

We can use `call/cc` to access the continuation k of the entire expression, and then explicitly apply this continuation to i with `throw`. This continuation application sends the value of i to the context of the whole expression. The corresponding DS expression thus reads:

$$\text{call/cc } \lambda k.g (f \lambda i.\text{throw } k i)$$

This is the result obtained by our extended DS transformation.

The rest of this paper formalizes this transformation.

1.4 Overview

Section 2 reviews our starting point: the DS transformation for pure CPS terms. The extended DS transformation is presented in Section 3. Section 4 proves the correctness of the DS transformations. In Section 5, we describe three applications. After a comparison with related work in Section 6, Section 7 concludes. Throughout, the figures are generated from runnable specifications.

$r \in \text{DRoot}$	— domain of DS λ -terms
$e \in \text{DExp}$	— domain of DS serious expressions
$t \in \text{DTriv}$	— domain of DS trivial expressions
$i \in \text{Ide}$	— domain of identifiers
$r ::= e$ $e ::= e_0 e_1 \mid t$ $t ::= i \mid \lambda i.r$	

Figure 1: Syntax of pure DS terms

2 Back to Direct Style I

Our starting point is the BNF of DS terms, shown in Figure 1. We refer to this language as the language of *pure DS terms* because it corresponds to the pure λ -calculus, without added control operators. Much as Reynolds [35], we distinguish between “serious” and “trivial” expressions. Evaluating a serious expression may create new contexts, whereas evaluating a trivial expression never does.

Figure 2 displays Danvy and Filinski’s CPS transformation [8]. The transformation is higher-order and is expressed using Nielson and Nielson’s two-level λ -calculus [32]. Underlined λ and $@$ respectively denote syntactic constructors for λ -abstractions and for (infix) applications. The CPS counterpart of a DS term r is obtained by $\mathcal{C}^{\text{DRoot}} \llbracket r \rrbracket$.

The language produced by this CPS transformation is described by the grammar displayed in Figure 3. We refer to this language as the language of *pure CPS terms* because it is derived from the CPS transformation of the language of pure DS terms. The language retains the structure of serious and trivial expressions. Here, a serious expression is evaluated in the scope of a continuation and a trivial expression denotes a value that is passed to a continuation.

Reflecting the fact that any DS expression occurs in only one context (the current context), a CPS expression accesses only one continuation (the current continuation). This property is captured in Figure 4, which specifies occurrence conditions over continuation identifiers in CPS terms.

Theorem 1 *For any DS term r , the CPS term $\mathcal{C}^{\text{DRoot}} \llbracket r \rrbracket$ satisfies the judgement $\vdash_{\text{ContIde}}^{\text{CRoot}} \mathcal{C}^{\text{DRoot}} \llbracket r \rrbracket$.*

$$\begin{aligned} \mathcal{C}^{\text{DRoot}} & : \text{DRoot} \rightarrow \text{CRoot} \\ \mathcal{C}^{\text{DRoot}} \llbracket e \rrbracket & = \underline{\lambda} k . \mathcal{C}^{\text{DExp}} \llbracket e \rrbracket \lambda t . k \underline{@} t \\ & \quad \text{— where } k \text{ is fresh.} \end{aligned}$$

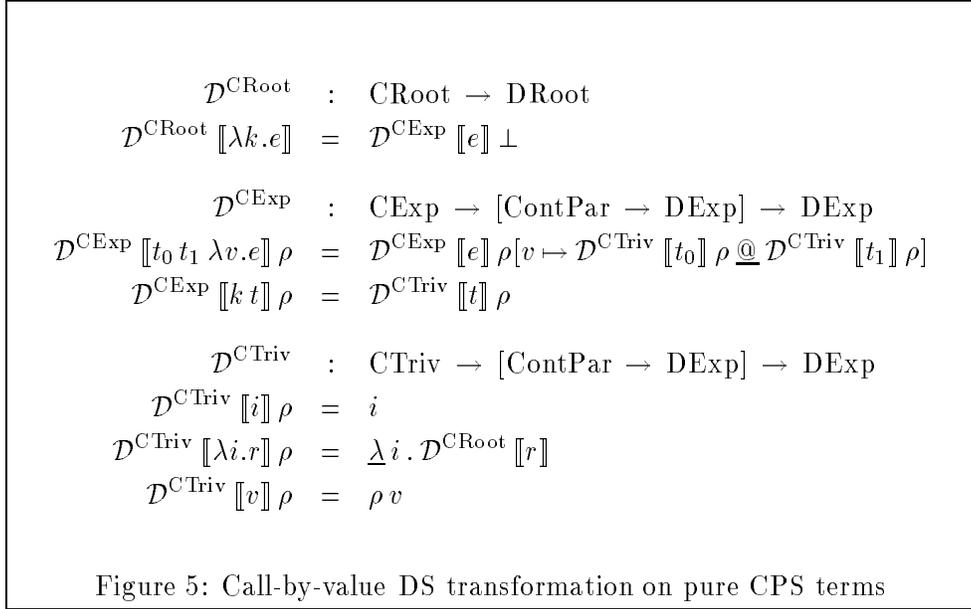
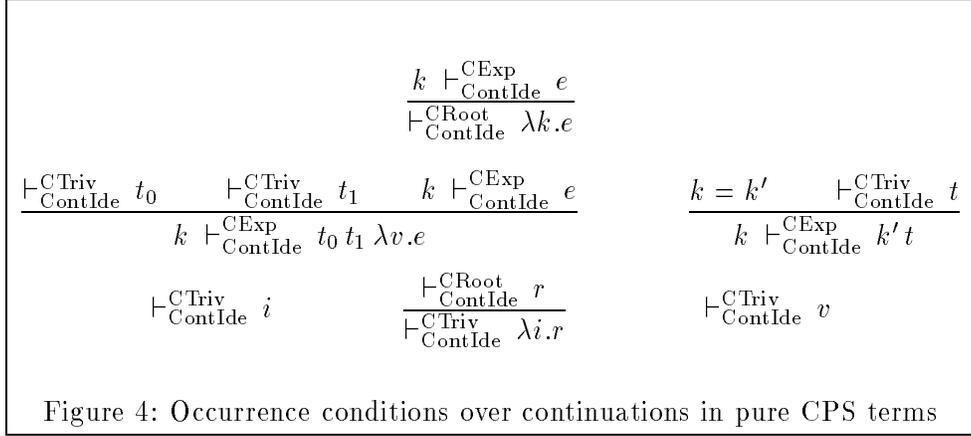
$$\begin{aligned} \mathcal{C}^{\text{DExp}} & : \text{DExp} \rightarrow [\text{CTriv} \rightarrow \text{CExp}] \rightarrow \text{CExp} \\ \mathcal{C}^{\text{DExp}} \llbracket e_0 e_1 \rrbracket \kappa & = \mathcal{C}^{\text{DExp}} \llbracket e_0 \rrbracket \lambda t_0 . \mathcal{C}^{\text{DExp}} \llbracket e_1 \rrbracket \lambda t_1 . t_0 \underline{@} t_1 \underline{@} \underline{\lambda} v . \kappa v \\ & \quad \text{— where } v \text{ is fresh.} \\ \mathcal{C}^{\text{DExp}} \llbracket t \rrbracket \kappa & = \kappa \mathcal{C}^{\text{DTriv}} \llbracket t \rrbracket \end{aligned}$$

$$\begin{aligned} \mathcal{C}^{\text{DTriv}} & : \text{DTriv} \rightarrow \text{CTriv} \\ \mathcal{C}^{\text{DTriv}} \llbracket i \rrbracket & = i \\ \mathcal{C}^{\text{DTriv}} \llbracket \lambda i . r \rrbracket & = \underline{\lambda} i . \mathcal{C}^{\text{DRoot}} \llbracket r \rrbracket \end{aligned}$$

Figure 2: Call-by-value CPS transformation on pure DS terms

$$\begin{aligned} r \in \text{CRoot} & \quad \text{— domain of CPS } \lambda\text{-terms} \\ e \in \text{CExp} & \quad \text{— domain of CPS serious expressions} \\ t \in \text{CTriv} & \quad \text{— domain of CPS trivial expressions} \\ i \in \text{Ide} & \quad \text{— domain of identifiers} \\ k \in \text{ContIde} & \quad \text{— domain of continuation identifiers,} \\ & \quad \text{disjoint from Ide} \\ v \in \text{ContPar} & \quad \text{— domain of continuation parameters,} \\ & \quad \text{disjoint from Ide and ContIde} \\ \\ r & ::= \lambda k . e \\ e & ::= t_0 t_1 \lambda v . e \mid k t \\ t & ::= i \mid \lambda i . r \mid v \end{aligned}$$

Figure 3: Syntax of CPS terms



Proof. Straightforward [7, 9, 27]. □

This property of CPS terms can also be captured by α -renaming all occurrences of continuation identifiers to be the same identifier k and by defining ContIde to be the singleton $\{k\}$ in Figure 3.¹

Figure 5 displays Danvy’s DS transformation [7]. The transformation is also expressed in the two-level λ -calculus. In particular, it uses a translation-time environment mapping parameters of continuations to DS applications. The empty environment is denoted \perp . Throughout this paper, we assume a call-by-value, left-to-right evaluation order. The correctness of this transformation is proved in Section 4.1.

3 Back to Direct Style II

As outlined in Section 1, we relax the conditions of Figure 4 to allow any continuation in scope to be applied, instead of only the current one. The relaxed conditions, which simply amount to checking that every continuation identifier is declared, are displayed in Figure 6. Now any CPS term r must satisfy the judgment $\emptyset \vdash_{\text{ContIde}}^{\text{CRoot}} r$, reflecting the fact that there can be no free continuation identifiers.

We want to extend the DS transformation conservatively. To this end, we want to treat ordinary continuation identifiers as usual. Figure 7 detects whether a continuation identifier k occurs only where permitted in the original CPS language, as characterized by Figure 4. We focus on the occurrence of a single continuation identifier, rather than on the structure of an entire term. Figure 7 thus omits the requirement that $k = k'$ in the continuation-application case, and replaces the test on a trivial term t by a check that k does not occur free in t . The declaration of a first-class continuation is thus any root expression that fails to satisfy the judgment $\vdash_{\text{NFC}}^{\text{CRoot}} r$.

We map the declaration of a first-class continuation into an occurrence of call/cc , and the first-class use of a continuation into an occurrence of throw . The resulting DS transformation is displayed in Figure 10. The BNF of the generated DS language is shown in Figure 9. Because a continuation can only be declared at the root of a CPS terms, and CPS roots are mapped into DS roots, call/cc can only occur at the root of a DS term. Because a continuation application does not itself take a continuation argument, a

¹This syntactic device is not uncommon. For example, Sabry and Felleisen use it in their work on reasoning about CPS programs [38].

$$\begin{array}{c}
\frac{\gamma \cup \{k\} \vdash_{\text{ContIde}}^{\text{CExp}} e}{\gamma \vdash_{\text{ContIde}}^{\text{CRoot}} \lambda k.e} \\
\\
\frac{\gamma \vdash_{\text{ContIde}}^{\text{CTriv}} t_0 \quad \gamma \vdash_{\text{ContIde}}^{\text{CTriv}} t_1 \quad \gamma \vdash_{\text{ContIde}}^{\text{CExp}} e}{\gamma \vdash_{\text{ContIde}}^{\text{CExp}} t_0 t_1 \lambda v.e} \quad \frac{k \in \gamma \quad \gamma \vdash_{\text{ContIde}}^{\text{CTriv}} t}{\gamma \vdash_{\text{ContIde}}^{\text{CExp}} k t} \\
\\
\gamma \vdash_{\text{ContIde}}^{\text{CTriv}} i \quad \frac{\gamma \vdash_{\text{ContIde}}^{\text{CRoot}} r}{\gamma \vdash_{\text{ContIde}}^{\text{CTriv}} \lambda i.r} \quad \gamma \vdash_{\text{ContIde}}^{\text{CTriv}} v
\end{array}$$

Figure 6: Occurrence conditions over (possibly first-class) continuations

$$\begin{array}{c}
\frac{k \vdash_{\text{NFC}}^{\text{CExp}} e}{\vdash_{\text{NFC}}^{\text{CRoot}} \lambda k.e} \\
\\
\frac{k \notin \text{FCI}^{\text{CTriv}} [t_0] \quad k \notin \text{FCI}^{\text{CTriv}} [t_1] \quad k \vdash_{\text{NFC}}^{\text{CExp}} e}{k \vdash_{\text{NFC}}^{\text{CExp}} t_0 t_1 \lambda v.e} \quad \frac{k \notin \text{FCI}^{\text{CTriv}} [t]}{k \vdash_{\text{NFC}}^{\text{CExp}} k' t}
\end{array}$$

Figure 7: Detection of non-first-class continuations

$$\begin{aligned}
\text{FCI}^{\text{CRoot}} [\lambda k.e] &= \text{FCI}^{\text{CExp}} [e] \setminus \{k\} \\
\text{FCI}^{\text{CExp}} [t_0 t_1 \lambda v.e] &= \text{FCI}^{\text{CTriv}} [t_0] \cup \text{FCI}^{\text{CTriv}} [t_1] \cup \text{FCI}^{\text{CExp}} [e] \\
\text{FCI}^{\text{CExp}} [k t] &= \text{FCI}^{\text{CTriv}} [t] \cup \{k\} \\
\text{FCI}^{\text{CTriv}} [i] &= \emptyset \\
\text{FCI}^{\text{CTriv}} [\lambda i.r] &= \text{FCI}^{\text{CRoot}} [r] \\
\text{FCI}^{\text{CTriv}} [v] &= \emptyset
\end{aligned}$$

Figure 8: Free continuation identifiers in a CPS term

$r \in \text{DRoot}$ — domain of DS λ -terms
 $r' \in \text{DRoot}'$ — domain of DS subroots
 $e \in \text{DExp}$ — domain of DS serious expressions
 $t \in \text{DTriv}$ — domain of DS trivial expressions
 $i \in \text{Ide}$ — domain of identifiers
 $k \in \text{ContIde}$ — domain of continuation identifiers, disjoint from Ide

$$\begin{aligned}
 r &::= r' \mid \text{call/cc } \lambda k.r' \\
 r' &::= e \mid \text{throw } k.e \\
 e &::= e_0.e_1 \mid t \\
 t &::= i \mid \lambda i.r
 \end{aligned}$$

Figure 9: Syntax of DS terms, including call/cc and throw

$$\begin{aligned}
 \mathcal{D}^{\text{CRoot}} &: \text{CRoot} \rightarrow \text{DRoot} \\
 \mathcal{D}^{\text{CRoot}} \llbracket \lambda k.e \rrbracket &= \begin{cases} \mathcal{D}^{\text{CExp}} \llbracket e \rrbracket \perp k & \text{if } k \vdash_{\text{NFC}}^{\text{CExp}} e \\ \text{call/cc } \lambda k. \mathcal{D}^{\text{CExp}} \llbracket e \rrbracket \perp k & \text{otherwise} \end{cases} \\
 \mathcal{D}^{\text{CExp}} &: \text{CExp} \rightarrow [\text{ContPar} \rightarrow \text{DExp}] \rightarrow \text{ContIde} \rightarrow \text{DExp} \\
 \mathcal{D}^{\text{CExp}} \llbracket t_0 t_1 \lambda v.e \rrbracket \rho k &= \mathcal{D}^{\text{CExp}} \llbracket e \rrbracket \rho [v \mapsto \mathcal{D}^{\text{CTriv}} \llbracket t_0 \rrbracket \rho \underline{\text{@}} \mathcal{D}^{\text{CTriv}} \llbracket t_1 \rrbracket \rho] k \\
 \mathcal{D}^{\text{CExp}} \llbracket k' t \rrbracket \rho k &= \begin{cases} \mathcal{D}^{\text{CTriv}} \llbracket t \rrbracket \rho & \text{if } k = k' \\ \underline{\text{throw}} k' (\mathcal{D}^{\text{CTriv}} \llbracket t \rrbracket \rho) & \text{otherwise} \end{cases} \\
 \mathcal{D}^{\text{CTriv}} &: \text{CTriv} \rightarrow [\text{ContPar} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
 \mathcal{D}^{\text{CTriv}} \llbracket i \rrbracket \rho &= i \\
 \mathcal{D}^{\text{CTriv}} \llbracket \lambda i.r \rrbracket \rho &= \lambda i. \mathcal{D}^{\text{CRoot}} \llbracket r \rrbracket \\
 \mathcal{D}^{\text{CTriv}} \llbracket v \rrbracket \rho &= \rho v
 \end{aligned}$$

Figure 10: Call-by-value DS transformation, including call/cc and throw

$$\begin{aligned}
\mathcal{C}^{\text{DRoot}} & : \text{DRoot} \rightarrow \text{CRoot} \\
\mathcal{C}^{\text{DRoot}} \llbracket r' \rrbracket & = \underline{\lambda} k . \mathcal{C}^{\text{DRoot}'} \llbracket r' \rrbracket k \quad \text{--- where } k \text{ is fresh.} \\
\mathcal{C}^{\text{DRoot}} \llbracket \text{call/cc } \lambda k . r' \rrbracket & = \underline{\lambda} k . \mathcal{C}^{\text{DRoot}'} \llbracket r' \rrbracket k \\
\mathcal{C}^{\text{DRoot}'} & : \text{DRoot}' \rightarrow \text{ContIde} \rightarrow \text{CExp} \\
\mathcal{C}^{\text{DRoot}'} \llbracket e \rrbracket k & = \mathcal{C}^{\text{DExp}} \llbracket e \rrbracket \lambda t . k \underline{\text{@}} t \\
\mathcal{C}^{\text{DRoot}'} \llbracket \text{throw } k' e \rrbracket k & = \mathcal{C}^{\text{DExp}} \llbracket e \rrbracket \lambda t . k' \underline{\text{@}} t \\
\mathcal{C}^{\text{DExp}} & : \text{DExp} \rightarrow [\text{CTriv} \rightarrow \text{CExp}] \rightarrow \text{CExp} \\
\mathcal{C}^{\text{DExp}} \llbracket e_0 e_1 \rrbracket \kappa & = \mathcal{C}^{\text{DExp}} \llbracket e_0 \rrbracket \lambda t_0 . \mathcal{C}^{\text{DExp}} \llbracket e_1 \rrbracket \lambda t_1 . t_0 \underline{\text{@}} t_1 \underline{\text{@}} \underline{\lambda} v . \kappa v \\
& \quad \text{--- where } v \text{ is fresh.} \\
\mathcal{C}^{\text{DExp}} \llbracket t \rrbracket \kappa & = \kappa (\mathcal{C}^{\text{DTriv}} \llbracket t \rrbracket) \\
\mathcal{C}^{\text{DTriv}} & : \text{DTriv} \rightarrow \text{CTriv} \\
\mathcal{C}^{\text{DTriv}} \llbracket i \rrbracket & = i \\
\mathcal{C}^{\text{DTriv}} \llbracket \lambda i . r \rrbracket & = \underline{\lambda} i . \mathcal{C}^{\text{DRoot}} \llbracket r \rrbracket
\end{aligned}$$

Figure 11: Call-by-value CPS transformation, including call/cc and throw

throw expression in DS does not create a new context. We explicitly identify such positions as the new class of DS subroots.

Figure 11 displays the traditional CPS transformation including call/cc and throw [3, 8, 15, 25, 40], restricted to the generated DS language. In Section 4.2, we prove that this CPS transformation and the extended DS transformation are inverses.

Both the CPS and the DS transformations scale up to more practical programming languages. The extension to a Scheme-like programming language that includes constants, primitive n -ary operators, uncurried functions, conditional expressions, and block structure is straightforward. The CPS transformation of such a language has been extensively studied [8, 15, 24, 26, 40]. The corresponding DS transformation is considered in detail in Lawall's PhD thesis [27]. The applications described in Section 5 make use of such a DS transformation.

4 Inverseness of the CPS and DS transformations

We now show that the CPS and DS transformations are inverses of each other. The argument is syntax-based. Because the CPS transformation is semantics-preserving [8, 13, 14, 33, 36, 39], this approach is sufficient to justify the correctness of the DS transformation.

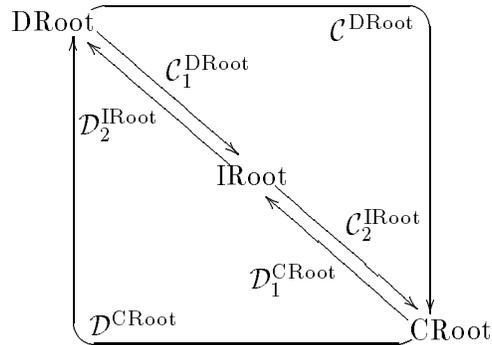
We begin by reviewing the proof that the pure CPS and DS transformations are inverses of each other. We then extend the approach to the extended transformations.

4.1 Back to Direct Style I

To prove that the CPS and DS transformations are inverses of each other, we stage both transformations via an intermediate language. CPS names intermediate results and sequentializes computations. Furthermore, it replaces the creation of a new context in DS by a continuation argument. The intermediate language names intermediate results and sequentializes computations while otherwise remaining in DS. Essentially, new contexts are identified, but continuations are not introduced.

The intermediate language is specified in Figure 12. It is essentially the λ -calculus with two special forms, `let` and `return`. Each `let` expression names an intermediate result, and since the intermediate language only allows flat `let` expressions, sequentiality is ensured. A `return` expression coerces a trivial expression into a serious one. This intermediate language is deliberately reminiscent of Moggi's monadic normal forms [7, 18, 30].

The following diagram summarizes the situation.



$r \in \text{IRoot}$ — domain of IS λ -terms
 $e \in \text{IExp}$ — domain of IS serious expressions
 $t \in \text{ITriv}$ — domain of IS trivial expressions
 $i \in \text{Ide}$ — domain of identifiers
 $v \in \text{LetPar}$ — domain of let parameters, disjoint from Ide

$r ::= e$
 $e ::= \text{let } v = t_0 t_1 \text{ in } e \mid \text{return } t$
 $t ::= i \mid \lambda i.r \mid v$

Figure 12: Syntax of pure intermediate-style terms

$\mathcal{C}_1^{\text{DRoot}} : \text{DRoot} \rightarrow \text{IRoot}$
 $\mathcal{C}_1^{\text{DRoot}}[e] = \mathcal{C}_1^{\text{DExp}}[e] \lambda t . \underline{\text{return}} t$

$\mathcal{C}_1^{\text{DExp}} : \text{DExp} \rightarrow [\text{ITriv} \rightarrow \text{IExp}] \rightarrow \text{IExp}$
 $\mathcal{C}_1^{\text{DExp}}[e_0 e_1] \kappa = \mathcal{C}_1^{\text{DExp}}[e_0] \lambda t_0 . \mathcal{C}_1^{\text{DExp}}[e_1] \lambda t_1 . \underline{\text{let}} v = t_0 \underline{@} t_1 \underline{\text{in}} \kappa v$
 — where v is fresh.

$\mathcal{C}_1^{\text{DExp}}[t] \kappa = \kappa \mathcal{C}_1^{\text{DTriv}}[t]$

$\mathcal{C}_1^{\text{DTriv}} : \text{DTriv} \rightarrow \text{ITriv}$
 $\mathcal{C}_1^{\text{DTriv}}[i] = i$
 $\mathcal{C}_1^{\text{DTriv}}[\lambda i.r] = \underline{\lambda} i . \mathcal{C}_1^{\text{DRoot}}[r]$

Figure 13: From direct style to intermediate style

$$\begin{aligned}
\mathcal{C}_2^{\text{IRoot}} & : \text{IRoot} \rightarrow \text{CRoot} \\
\mathcal{C}_2^{\text{IRoot}}[e] & = \underline{\lambda} k . \mathcal{C}_2^{\text{IExp}}[e] k \quad \text{--- where } k \text{ is fresh.} \\
\mathcal{C}_2^{\text{IExp}} & : \text{IExp} \rightarrow \text{ContIde} \rightarrow \text{CExp} \\
\mathcal{C}_2^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] k & = \mathcal{C}_2^{\text{ITriv}}[t_0] \ @ \ \mathcal{C}_2^{\text{ITriv}}[t_1] \ @ \ \underline{\lambda} v . \mathcal{C}_2^{\text{IExp}}[e] k \\
\mathcal{C}_2^{\text{IExp}}[\text{return } t] k & = k \ @ \ \mathcal{C}_2^{\text{ITriv}}[t] \\
\mathcal{C}_2^{\text{ITriv}} & : \text{ITriv} \rightarrow \text{CTriv} \\
\mathcal{C}_2^{\text{ITriv}}[i] & = i \\
\mathcal{C}_2^{\text{ITriv}}[\underline{\lambda} i . r] & = \underline{\lambda} i . \mathcal{C}_2^{\text{IRoot}}[r] \\
\mathcal{C}_2^{\text{ITriv}}[v] & = v
\end{aligned}$$

Figure 14: From intermediate style to continuation-passing style (continuation introduction)

$$\begin{aligned}
\mathcal{D}_1^{\text{CRoot}} & : \text{CRoot} \rightarrow \text{IRoot} \\
\mathcal{D}_1^{\text{CRoot}}[\underline{\lambda} k . e] & = \mathcal{D}_1^{\text{CExp}}[e] \\
\mathcal{D}_1^{\text{CExp}} & : \text{CExp} \rightarrow \text{IExp} \\
\mathcal{D}_1^{\text{CExp}}[t_0 \ t_1 \ \underline{\lambda} v . e] & = \underline{\text{let}} \ v = \mathcal{D}_1^{\text{CTriv}}[t_0] \ @ \ \mathcal{D}_1^{\text{CTriv}}[t_1] \ \underline{\text{in}} \ \mathcal{D}_1^{\text{CExp}}[e] \\
\mathcal{D}_1^{\text{CExp}}[k \ t] & = \underline{\text{return}} \ \mathcal{D}_1^{\text{CTriv}}[t] \\
\mathcal{D}_1^{\text{CTriv}} & : \text{CTriv} \rightarrow \text{ITriv} \\
\mathcal{D}_1^{\text{CTriv}}[i] & = i \\
\mathcal{D}_1^{\text{CTriv}}[\underline{\lambda} i . r] & = \underline{\lambda} i . \mathcal{D}_1^{\text{CRoot}}[r] \\
\mathcal{D}_1^{\text{CTriv}}[v] & = v
\end{aligned}$$

Figure 15: From continuation-passing style to intermediate style (continuation elimination)

$$\begin{aligned}
\mathcal{D}_2^{\text{IRoot}} & : \text{IRoot} \rightarrow \text{DRoot} \\
\mathcal{D}_2^{\text{IRoot}}[e] & = \mathcal{D}_2^{\text{IExp}}[e] \perp \\
\mathcal{D}_2^{\text{IExp}} & : \text{IExp} \rightarrow [\text{LetPar} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
\mathcal{D}_2^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] \rho & = \mathcal{D}_2^{\text{IExp}}[e] \rho[v \mapsto \mathcal{D}_2^{\text{ITriv}}[t_0] \rho \ @ \ \mathcal{D}_2^{\text{ITriv}}[t_1] \rho] \\
\mathcal{D}_2^{\text{IExp}}[\text{return } t] \rho & = \mathcal{D}_2^{\text{ITriv}}[t] \rho \\
\mathcal{D}_2^{\text{ITriv}} & : \text{ITriv} \rightarrow [\text{LetPar} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
\mathcal{D}_2^{\text{ITriv}}[i] \rho & = i \\
\mathcal{D}_2^{\text{ITriv}}[\lambda i . r] \rho & = \lambda i . \mathcal{D}_2^{\text{IRoot}}[r] \\
\mathcal{D}_2^{\text{ITriv}}[v] \rho & = \rho v
\end{aligned}$$

Figure 16: From intermediate style to direct style (let unfolding)

Lemma 1 *The CPS and DS transformation can be staged through the intermediate language of Figure 12.*

Proof. Figure 13 displays the encoding \mathcal{C}_1 of the DS language into the intermediate language. Figure 14 displays the transformation \mathcal{C}_2 from the intermediate language forth to CPS. Figure 15 displays the encoding \mathcal{D}_1 of the CPS language into the intermediate language. Figure 16 displays the transformation \mathcal{D}_2 from the intermediate language back to direct style.

Composing $\mathcal{C}_1^{\text{DRoot}}$ and $\mathcal{C}_2^{\text{IRoot}}$ yields $\mathcal{C}^{\text{DRoot}}$; composing $\mathcal{D}_1^{\text{CRoot}}$ and $\mathcal{D}_2^{\text{IRoot}}$ yields $\mathcal{D}^{\text{CRoot}}$ [7, 27]. \square

Lemma 2 $\mathcal{C}_1^{\text{DRoot}}$ and $\mathcal{D}_2^{\text{IRoot}}$ are inverses of each other, modulo renaming of bound variables.

Proof. $\mathcal{C}_1^{\text{DRoot}}$ introduces let expressions and flattens them. $\mathcal{D}_2^{\text{IRoot}}$ unfolds these let expressions. $\mathcal{D}_2^{\text{IRoot}}$ is thus a left inverse of $\mathcal{C}_1^{\text{DRoot}}$ [27].

$\mathcal{C}_1^{\text{DRoot}}$ gives rise to occurrence conditions that are characteristic of left-to-right, call-by-value evaluation. These syntactic conditions are necessary to prove that $\mathcal{D}_2^{\text{IRoot}}$ is a right inverse of $\mathcal{C}_1^{\text{DRoot}}$ [7, 9, 27]. \square

Lemma 3 $\mathcal{D}_1^{\text{CRoot}}$ and $\mathcal{C}_2^{\text{IRoot}}$ are inverses of each other, modulo renaming of bound variables.

Proof. Straightforward [28]. □

Theorem 2 *The CPS and DS transformations are inverses of each other, modulo renaming of bound variables.*

Proof. A consequence of Lemmas 1, 2, and 3. □

4.2 Back to Direct Style II

To prove that the extended CPS and DS transformations are inverses of each other, we extend the proof of Theorem 2. To this end, we extend the intermediate language of Figure 12 with call/cc and throw, as shown in Figure 17. We again stage the CPS and DS transformations through this intermediate language. Control operations are translated when continuations are introduced and eliminated, *i.e.*, in the transformations between IS and CPS. Our investigation of inverseness thus focuses on these transformations.

Figure 18 displays the encoding \mathcal{C}_1 of the DS language into the intermediate language. Figure 19 displays the transformation \mathcal{C}_2 from the intermediate language forth to CPS. Figure 20 displays the encoding \mathcal{D}_1 of the CPS language into the intermediate language. Figure 21 displays the transformation \mathcal{D}_2 from the intermediate language back to direct style.

The transformation from CPS to IS and back to CPS, $\mathcal{C}_2^{\text{IRoot}} \circ \mathcal{D}_1^{\text{CRoot}}$, is straightforwardly the identity transformation. Whenever $\mathcal{D}_1^{\text{CRoot}}$ decides whether to introduce a control operator, $\mathcal{C}_2^{\text{IRoot}}$ translates both possibilities into an identical CPS term. The other cases follow the proof of Lemma 3.

The analysis of the transformation from IS to CPS and back to IS, $\mathcal{D}_1^{\text{CRoot}} \circ \mathcal{C}_2^{\text{IRoot}}$, is more complicated, because control operations are not explicit in CPS. The composition of these transformations applied to IRoot expressions and to return and throw expressions is shown below. The other cases are straightforward.

$$\begin{aligned}
& \mathcal{D}_1^{\text{CRoot}}[\mathcal{C}_2^{\text{IRoot}}[e]] \\
&= \mathcal{D}_1^{\text{CRoot}}[\lambda k . \mathcal{C}_2^{\text{IExp}}[e] k] \quad \text{— where } k \text{ is fresh.} \\
&= \begin{cases} \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[e] k] k & \text{if } k \vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] k \\ \text{call/cc } \lambda k . \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[e] k] k & \text{otherwise} \end{cases}
\end{aligned}$$

$r \in \text{IRoot}$ — domain of IS λ -terms
 $e \in \text{IExp}$ — domain of IS serious expressions
 $t \in \text{ITriv}$ — domain of IS trivial expressions
 $i \in \text{Ide}$ — domain of identifiers
 $v \in \text{LetPar}$ — domain of let parameters, disjoint from Ide

$$\begin{aligned}
 r &::= e \mid \text{call/cc } \lambda k.e \\
 e &::= \text{let } v = t_0 t_1 \text{ in } e \mid \text{return } t \mid \text{throw } k t \\
 t &::= i \mid \lambda i.r \mid v
 \end{aligned}$$

Figure 17: Syntax of intermediate-style terms, including call/cc and throw

$$\begin{aligned}
 \mathcal{C}_1^{\text{DRoot}} &: \text{DRoot} \rightarrow \text{IRoot} \\
 \mathcal{C}_1^{\text{DRoot}}[[r']] &= \mathcal{C}_1^{\text{DRoot}'}[[r']] \\
 \mathcal{C}_1^{\text{DRoot}}[[\text{call/cc } \lambda k.r']] &= \underline{\text{call/cc}} \lambda k . \mathcal{C}_1^{\text{DRoot}'}[[r']] \\
 \\
 \mathcal{C}_1^{\text{DRoot}'} &: \text{DRoot}' \rightarrow \text{IExp} \\
 \mathcal{C}_1^{\text{DRoot}'}[[e]] &= \mathcal{C}_1^{\text{DExp}}[[e]] \lambda t . \underline{\text{return}} t \\
 \mathcal{C}_1^{\text{DRoot}'}[[\text{throw } k e]] &= \mathcal{C}_1^{\text{DExp}}[[e]] \lambda t . \underline{\text{throw}} k t \\
 \\
 \mathcal{C}_1^{\text{DExp}} &: \text{DExp} \rightarrow [\text{ITriv} \rightarrow \text{IExp}] \rightarrow \text{IExp} \\
 \mathcal{C}_1^{\text{DExp}}[[e_0 e_1]] \kappa &= \mathcal{C}_1^{\text{DExp}}[[e_0]] \lambda t_0 . \mathcal{C}_1^{\text{DExp}}[[e_1]] \lambda t_1 . \underline{\text{let}} v = t_0 \underline{\text{@}} t_1 \underline{\text{in}} \kappa v \\
 &\quad \text{— where } v \text{ is fresh.} \\
 \mathcal{C}_1^{\text{DExp}}[[t]] \kappa &= \kappa \mathcal{C}_1^{\text{DTriv}}[[t]] \\
 \\
 \mathcal{C}_1^{\text{DTriv}} &: \text{DTriv} \rightarrow \text{ITriv} \\
 \mathcal{C}_1^{\text{DTriv}}[[i]] &= i \\
 \mathcal{C}_1^{\text{DTriv}}[[\lambda i.r]] &= \underline{\lambda} i . \mathcal{C}_1^{\text{DRoot}}[[r]]
 \end{aligned}$$

Figure 18: From direct style to intermediate style, including call/cc and throw

$$\begin{aligned}
\mathcal{C}_2^{\text{IRoot}} &: \text{IRoot} \rightarrow \text{CRoot} \\
\mathcal{C}_2^{\text{IRoot}}[e] &= \underline{\lambda} k . \mathcal{C}_2^{\text{IExp}}[e] k \quad \text{--- where } k \text{ is fresh.} \\
\mathcal{C}_2^{\text{IRoot}}[\text{call/cc } \lambda k . e] &= \underline{\lambda} k . \mathcal{C}_2^{\text{IExp}}[e] k \\
\\
\mathcal{C}_2^{\text{IExp}} &: \text{IExp} \rightarrow \text{ContIde} \rightarrow \text{CExp} \\
\mathcal{C}_2^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] k &= \mathcal{C}_2^{\text{ITriv}}[t_0] \ @ \ \mathcal{C}_2^{\text{ITriv}}[t_1] \ @ \ \underline{\lambda} v . \mathcal{C}_2^{\text{IExp}}[e] k \\
\mathcal{C}_2^{\text{IExp}}[\text{return } t] k &= k \ @ \ \mathcal{C}_2^{\text{ITriv}}[t] \\
\mathcal{C}_2^{\text{IExp}}[\text{throw } k' \ t] k &= k' \ @ \ \mathcal{C}_2^{\text{ITriv}}[t] \\
\\
\mathcal{C}_2^{\text{ITriv}} &: \text{ITriv} \rightarrow \text{CTriv} \\
\mathcal{C}_2^{\text{ITriv}}[i] &= i \\
\mathcal{C}_2^{\text{ITriv}}[\lambda i . r] &= \underline{\lambda} i . \mathcal{C}_2^{\text{IRoot}}[r] \\
\mathcal{C}_2^{\text{ITriv}}[v] &= v
\end{aligned}$$

Figure 19: From intermediate style to continuation-passing style, including call/cc and throw

$$\begin{aligned}
\mathcal{D}_1^{\text{CRoot}} &: \text{CRoot} \rightarrow \text{IRoot} \\
\mathcal{D}_1^{\text{CRoot}}[\lambda k . e] &= \begin{cases} \mathcal{D}_1^{\text{CExp}}[e] k & \text{if } k \vdash_{\text{NFC}}^{\text{CExp}} e \\ \text{call/cc } \underline{\lambda} k . \mathcal{D}_1^{\text{CExp}}[e] k & \text{otherwise} \end{cases} \\
\\
\mathcal{D}_1^{\text{CExp}} &: \text{CExp} \rightarrow \text{ContIde} \rightarrow \text{IExp} \\
\mathcal{D}_1^{\text{CExp}}[t_0 \ t_1 \ \lambda v . e] k &= \underline{\text{let}} \ v = \mathcal{D}_1^{\text{CTriv}}[t_0] \ @ \ \mathcal{D}_1^{\text{CTriv}}[t_1] \ \underline{\text{in}} \ \mathcal{D}_1^{\text{CExp}}[e] k \\
\mathcal{D}_1^{\text{CExp}}[k' \ t] k &= \begin{cases} \underline{\text{return}} \ \mathcal{D}_1^{\text{CTriv}}[t] & \text{if } k = k' \\ \underline{\text{throw}} \ k' \ \mathcal{D}_1^{\text{CTriv}}[t] & \text{otherwise} \end{cases} \\
\\
\mathcal{D}_1^{\text{CTriv}} &: \text{CTriv} \rightarrow \text{ITriv} \\
\mathcal{D}_1^{\text{CTriv}}[i] &= i \\
\mathcal{D}_1^{\text{CTriv}}[\lambda i . r] &= \underline{\lambda} i . \mathcal{D}_1^{\text{CRoot}}[r] \\
\mathcal{D}_1^{\text{CTriv}}[v] &= v
\end{aligned}$$

Figure 20: From continuation-passing style to intermediate style, including call/cc and throw

$$\begin{aligned}
\mathcal{D}_2^{\text{IRoot}} & : \text{IRoot} \rightarrow \text{DRoot} \\
\mathcal{D}_2^{\text{IRoot}}[e] & = \mathcal{D}_2^{\text{IExp}}[e] \perp \\
\mathcal{D}_2^{\text{IRoot}}[\text{call/cc } \lambda k . e] & = \underline{\text{call/cc}} \lambda k . \mathcal{D}_2^{\text{IExp}}[e] \perp \\
\\
\mathcal{D}_2^{\text{IExp}} & : \text{IExp} \rightarrow [\text{LetPar} \rightarrow \text{DExp}] \rightarrow \text{DRoot}' \\
\mathcal{D}_2^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] \rho & = \mathcal{D}_2^{\text{IExp}}[e] \rho [v \mapsto \mathcal{D}_2^{\text{ITriv}}[t_0] \rho \ @ \ \mathcal{D}_2^{\text{ITriv}}[t_1] \rho] \\
\mathcal{D}_2^{\text{IExp}}[\text{return } t] \rho & = \mathcal{D}_2^{\text{ITriv}}[t] \rho \\
\mathcal{D}_2^{\text{IExp}}[\text{throw } k' \ t] \rho & = \underline{\text{throw}} \ k' \ \mathcal{D}_2^{\text{ITriv}}[t] \rho \\
\\
\mathcal{D}_2^{\text{ITriv}} & : \text{ITriv} \rightarrow [\text{LetPar} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
\mathcal{D}_2^{\text{ITriv}}[i] \rho & = i \\
\mathcal{D}_2^{\text{ITriv}}[\lambda i . r] \rho & = \lambda i . \mathcal{D}_2^{\text{IRoot}}[r] \\
\mathcal{D}_2^{\text{ITriv}}[v] \rho & = \rho \ v
\end{aligned}$$

Figure 21: From intermediate style to direct style, including call/cc and throw

$$\begin{aligned}
& \mathcal{D}_1^{\text{CRoot}}[\mathcal{C}_2^{\text{IRoot}}[\text{call/cc } \lambda k . e]] \\
& = \mathcal{D}_1^{\text{CRoot}}[\lambda k . \mathcal{C}_2^{\text{IExp}}[e] \ k] \\
& = \begin{cases} \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[e] \ k] \ k & \text{if } k \vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] \ k \\ \text{call/cc } \lambda k . \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[e] \ k] \ k & \text{otherwise} \end{cases} \\
\\
& \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[\text{return } t] \ k] \ k \\
& = \mathcal{D}_1^{\text{CExp}}[k \ @ \ \mathcal{C}_2^{\text{ITriv}}[t]] \ k \\
& = \text{return } \mathcal{D}_1^{\text{CTriv}}[\mathcal{C}_2^{\text{ITriv}}[t]] \\
\\
& \mathcal{D}_1^{\text{CExp}}[\mathcal{C}_2^{\text{IExp}}[\text{throw } k' \ t] \ k] \ k \\
& = \mathcal{D}_1^{\text{CExp}}[k' \ @ \ \mathcal{C}_2^{\text{ITriv}}[t]] \ k \\
& = \begin{cases} \text{return } \mathcal{D}_1^{\text{CTriv}}[\mathcal{C}_2^{\text{ITriv}}[t]] & \text{if } k = k' \\ \text{throw } k' \ (\mathcal{D}_1^{\text{CTriv}}[\mathcal{C}_2^{\text{ITriv}}[t]]) & \text{otherwise} \end{cases}
\end{aligned}$$

Inverseness thus depends on the following properties:

- When the root has the form e , we require

$$k \vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] k$$

for a fresh k .

- When the root has the form $\text{call/cc } \lambda k.e$, we require

$$k \not\vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] k$$

for the same k .

- $\text{throw } k't$ should occur only when the continuation identifier argument to $\mathcal{D}_1^{\text{CExp}}$ is k , for $k \neq k'$.

The first case is ensured by the following lemma:

Lemma 4 *For fresh k , $k \vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] k$.*

Proof. Straightforward. □

$k \vdash_{\text{NFC}}^{\text{CExp}} e$ is violated only when k occurs in a trivial subexpression of the CPS term e . Via $\mathcal{C}_2^{\text{IRoot}}$, $k \vdash_{\text{NFC}}^{\text{CExp}} \mathcal{C}_2^{\text{IExp}}[e] k$ is violated when k occurs in a trivial subexpression of the IS term e , but not when it occurs in the body of arbitrarily many nested let expressions. Thus $\mathcal{C}_2^{\text{IRoot}}$ recreates the call/cc expression only in the former case.

The continuation identifier argument used by $\mathcal{D}_1^{\text{CExp}}$ to translate a CPS expression, is always the same as the continuation identifier argument used by $\mathcal{C}_2^{\text{IExp}}$ to create the CPS expression. Thus, if the continuation identifier argument to the translation of $\text{throw } k't$ is different from k' , the resulting continuation application is translated by $\mathcal{D}_1^{\text{CExp}}$ back into a throw expression. $\mathcal{C}_2^{\text{IExp}}$ simply propagates a continuation identifier from the point of declaration into the translation of the bodies of nested let expressions. Thus the condition that all continuation identifiers occur in trivial IS subexpressions ensures that all throw expressions are reconstructed as well.

These observations are summarized by the following equations:

$$\text{call/cc } \lambda k.\text{let } \dots \text{ in throw } k e \equiv \text{call/cc } \lambda k.\text{let } \dots \text{ in } e \quad (1)$$

$$\text{call/cc } \lambda k.e \equiv e \quad \text{— whenever } k \notin \text{FV}(e). \quad (2)$$

where the free-variable function FV is extended to the equivalence classes $[e]$ determined by these equations as follows:

$$k \in \text{FV}([e]) \Leftrightarrow \forall e', [e'] = [e] \Rightarrow k \in \text{FV}(e')$$

Through \mathcal{D}_2 , these equations are expressed more naturally in the DS language:

$$\text{call/cc } \lambda k. \text{throw } k \ e \equiv \text{call/cc } \lambda k. e \quad (3)$$

$$\text{call/cc } \lambda k. e \equiv e \quad \text{— whenever } k \notin \text{FV}([e]). \quad (4)$$

These equations express the idea that all the captured continuations are eventually used, albeit not immediately.

We are now ready to prove inverseness, following the strategy for the pure language.

Lemma 5 *The CPS and DS transformation can be staged through the intermediate language of Figure 17.*

Proof. Straightforward extension of Lemma 1 to the transformations in Figures 18, 19, 20, and 21. \square

Lemma 6 $\mathcal{C}_1^{\text{DRoot}}$ and $\mathcal{D}_2^{\text{IRoot}}$ are inverses of each other, modulo renaming of bound variables.

Proof. Straightforward extension of Lemma 2. \square

Lemma 7 $\mathcal{D}_1^{\text{CRoot}}$ and $\mathcal{C}_2^{\text{IRoot}}$ are inverses of each other, modulo renaming of bound variables and Equations 1 and 2.

Proof. Straightforward extension of Lemma 3, and as discussed above. \square

Theorem 3 *The CPS and DS transformations are inverses of each other, modulo renaming of bound variables and Equations 3 and 4.*

Proof. A consequence of Lemmas 5, 6, and 7. \square

5 Applications

We apply the direct-style transformation to three examples: a definitional interpreter for Scheme, partial evaluation of programs with coroutines, and normalization of simply typed programs extracted from classical proofs.

5.1 An interpreter for Scheme 84 (revisited)

Handwritten CPS programs are notoriously difficult to read. It is not easily apparent in a CPS program when continuations implement control effects, rather than normal procedure calls and returns. Converting handwritten CPS programs back to direct style can clarify the structure of such programs.

As an example of this approach, we consider the CPS interpreter for Scheme 84 presented by Haynes, Friedman, and Wand in the proceedings of LFP’84 [19, Fig. 1, p. 295]. Our DS transformer maps this interpreter to the natural direct-style specification of Scheme. The result (see Figures 25, 26, and 27 in appendix) uses `call/cc` to implement `call/cc`. Otherwise it looks like any other meta-circular Scheme interpreter. CPS-transforming it yields the original CPS interpreter. Therefore, the only reason to write the original interpreter in CPS was to specify `call/cc`.

5.2 Partial Evaluation

We can use the DS transformation in conjunction with the CPS transformation to carry out the automatic specialization of programs containing control operators. As an example, we specialize the classical samefringe program for binary trees with respect to one binary tree. We express this program using `call/cc` but without side-effects, based on the detach model of coroutines [6, 19]. The program is first transformed into continuation-passing style; it is then specialized. The result is then transformed back into direct style.

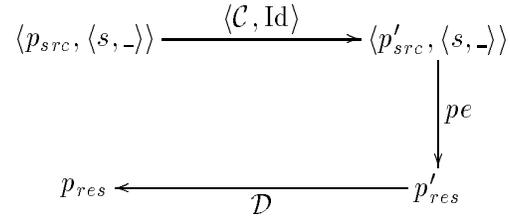
5.2.1 Partial evaluation

Partial evaluation is a semantics-based program transformation technique aimed at specializing a “source” program p_{src} with respect to a “static” part s of its input data [5, 22]. Partial evaluation produces a “residual” program p_{res} . The programs p_{src} and p_{res} are related as follows. Running p_{res} on the “dynamic” part d of the input data produces the same result as running p_{src} on both s and d (but usually running p_{res} is more efficient). This is captured in the following equations that paraphrase Kleene’s S_n^m -theorem.

$$\begin{cases} \text{run } pe \langle p_{src}, \langle s, - \rangle \rangle & = p_{res} \\ \text{run } p_{res} \langle -, d \rangle & = \text{run } p_{src} \langle s, d \rangle \end{cases}$$

In the first equation, pe denotes a partial evaluator. Of course, these equations only hold for terminating programs p_{src} and p_{res} , and if partial evaluation terminates.

Short of a source-level partial evaluator able to handle control operators directly [29], if we want to specialize a program involving `call/cc`, it is natural first to transform it into CPS (with \mathcal{C}), then to specialize it using a higher-order partial evaluator, and finally to transform the residual program into direct style (with \mathcal{D}). This approach is captured in the following diagram.



We are using Consel’s partial evaluator Schism [4].

5.2.2 The experiment

Figures 23 and 22 display the source program and its data structures.² We automatically transform this program into CPS. Schism automatically specializes it. We automatically transform the result into DS. Figure 24 displays a slightly pretty-printed version of the result (local variables have been renamed).

5.2.3 Assessment

As a whole, the static coroutine has been executed statically. Its computational content has been entirely inlined in the main procedure, yielding an iterative-looking residual program — though in fact, the dynamic binary tree is still traversed recursively. The resulting program uses one separate coroutine to traverse the dynamic binary tree.

One could argue that the resulting program should not use any coroutine at all, but this would require a more radical program manipulation. Such a transformation would involve a global Eureka step, as in Burstall and Darlington’s framework [2]. Regardless, if we want to specialize an n -ary samefringe program with respect to part of its input, the residual program would naturally be expressed in coroutine style.

²We use Schism syntactic facilities for declaring and using data types (more precisely: constructor names and their arities). These facilities are not standard in Scheme, but they can be easily defined as macros.

```

(define main
  (lambda (bt1 bt2)          ;;; Binary-Tree * Binary-Tree -> Bool
    (skim (initialize bt1) (initialize bt2))))

(define skim
  (lambda (d1 d2)          ;;; Data * Data -> Bool
    (caseType d1
      [(Next v1 k1)
       (caseType d2
         [(Next v2 k2)
          (and (equal? v1 v2)
               (skim (resume k1) (resume k2)))]
         [(Over) #f])]
      [(Over)
       (caseType d2
         [(Next v2 k2) #f]
         [(Over) #t])]))))

(define initialize
  (lambda (bt)          ;;; Binary-Tree -> Data
    (call/cc (lambda (k)
                ((defoliate bt (lambda (v) (throw k v))) (Over))))))

(define resume
  (lambda (c)          ;;; Cont -> Ans
    (call/cc (lambda (k)
                (c (lambda (v) (throw k v)))))))

(define defoliate
  (lambda (bt k)          ;;; Binary-Tree * Cont -> Ans
    (caseType bt
      [(Pair left right)
       (defoliate right (defoliate left k))]
      [(Leaf value)
       (call/cc (lambda (kp)
                   (k (Next value (lambda (v) (throw kp v))))))]))))

```

Figure 22: Control structures for the Samefringe program

```

(defineType Binary-Tree
  (Pair left right)
  (Leaf value))

(defineType Data
  (Next value continuation)
  (Over))

```

Figure 23: Data structures for the Samefringe program

```

;;; for all bt2,
;;; (main0 bt2) == (main (Pair (Pair (Leaf 0) (Leaf 1))
;;;                       (Pair (Leaf 2) (Leaf 3))))
;;;                       bt2)

(define main0
  (lambda (bt2)
    ;;; Binary-Tree -> Bool
    (caseType (initialize bt2)
      [(Next 11 k1)
       (and (equal? 0 11)
            (caseType (resume k1)
                      [(Next 12 k2)
                       (and (equal? 1 12)
                            (caseType (resume k2)
                                      [(Next 13 k3)
                                       (and (equal? 2 13)
                                            (caseType (resume k3)
                                                      [(Next 14 k4)
                                                       (and (equal? 3 14)
                                                            (caseType (resume k4)
                                                                      [(Next 15 k5) #f]
                                                                      [(Over) #t]))])
                                                       [(Over) #f]))])
                                       [(Over) #f]))])
                       [(Over) #f]))])
      [(Over) #f]))])

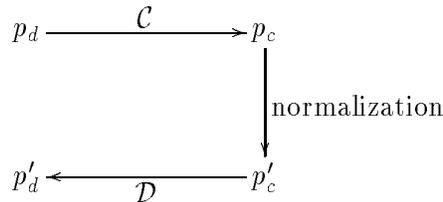
```

Figure 24: Specialized version of the Samefringe program

In any case, this experiment illustrates a first: the successful specialization of programs involving operations over control.

5.3 Program extraction from classical proofs

Programs extracted from classical proofs typically have many occurrences of call/cc [31]. To normalize such programs, we must extend a normalizer for the simply typed λ -calculus to handle call/cc. This, however, is unnecessary, given the direct-style transformation. As suggested in Section 5.2, one can instead CPS-transform the extracted program, normalize the resulting CPS program, and map the normalized program back to direct style.



6 Related Work

Naturally, this paper relies on Fischer’s, Plotkin’s, and Steele’s fundamental work on CPS [14, 33, 37, 40]. The CPS transformation described by Danvy and Filinski [8] is the point of reference for Sections 2 and 3. In an earlier work [7], Danvy developed the CPS-to-DS transformation described in Section 2. In her PhD thesis [27], Lawall proved the syntactic relationship between the CPS and DS transformations for a Scheme-like language, following the strategy described here.

In their work on reasoning about CPS programs [38, 39], Sabry and Felleisen have developed an “unCPS” transformation that is reminiscent of the DS transformation. They consider normalized Fischer-style CPS programs (*i.e.*, curried CPS programs with continuations occurring first), and formalize the DS counterpart of CPS simplifications. They transform each declaration of a continuation into a call/cc expression, and simplify the resulting term by eliminating all useless occurrences of call/cc. In contrast, the goal of our work is to consider DS and CPS programs that one could realistically write by hand, and our introduction of call/cc expressions is more sparse.

The detection of first-class continuations in Section 3 is purely syntactic and thus contrasts with Jouvelot and Gifford’s or with Deutsch’s more ambitious semantic analyses of programs with control effects [10, 23].

7 Conclusion

We have extended the DS transformation to handle first-class continuations. This extension is conservative and relies on a continuation-occurrence analysis that detects first-class occurrences of continuations in a CPS term. The DS and the CPS transformations widen the class of programs that can be manipulated on a semantic basis.

Just like the CPS transformation, the DS transformation can be extended with sequencing and with other computational effects such as assignments, destructive updates, and i/o [25, 40]. Sequencing is CPS-transformed with a continuation that does not use its parameter; conversely, a continuation whose parameter is not used can be mapped back to a sequence expression. The CPS transformation of side-effecting primitive operations is naturally achieved with continuation-passing versions of the primitive operators. These make it straightforward to go back to direct style (see the primitive operators `store-c!` and `store!` in Figures 25 and 27).

Based on the Curry-Howard isomorphism, the CPS transformation has been related to transformations on representations of proofs [17, 31]. The CPS transformation on types corresponds to the double-negation translation (defining the final domain of answers as falsity), and call/cc corresponds to Peirce’s law. By the same token, the DS transformation of CPS terms into DS terms with `call/cc` should have an interpretation in proof theory. We leave this aspect for a future work.

Over the last few years, many new control operators have emerged [8, 11, 12, 20, 34, 42]. If CPS is to be used as a unifying framework to specify and relate them, it must be possible to shift back and forth between programs using these operators and purely functional programs. Therefore it is useful to establish a sound understanding of the DS transformation and its relation to the CPS transformation.

Acknowledgments

Daniel Friedman and Harry Mairson provided doctoral and post-doctoral support to the second author. Andrzej Filinski, Karoline Malmkjær, and the LFP92 reviewers commented on earlier versions of this paper.

The diagrams of Sections 4 and 5 were drawn with Kristoffer Rose's Xy-pic package.

A Two Interpreters for Scheme 84

The language defined in Figures 25 and 26 offers constant expressions, identifiers, lexically scoped first-class functions, conditional expressions, lexical assignments, sequencing, call/cc, and applications. The specification is properly tail-recursive. (NB: the side-effecting primitive operator `store-c!` is in CPS.)

The language defined in Figure 27 is the same as in Figures 25 and 26. The specification is properly tail-recursive as well. Because each branch of a case expression is in tail-position with respect to the entire case expression, the `call/cc` expression in the `call/cc` branch could also be located around the case expression. Since the captured continuation is only used in one of the conditional branches, however, `call/cc` is more naturally located there. Notice how the unused continuation parameter is accounted for with a `begin` expression, in the definition of `evaluate-all`. (NB: the side-effecting primitive operator `store!` is in DS.)

References

- [1] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.
- [2] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of ACM*, 24(1):44–67, 1977.
- [3] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.
- [4] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

```

(lambda (k)
  (letrec ([meaning          ;;; Exp * Env * [Val -> Ans] -> Ans
           (lambda (e r k)
             (case (type-of-expression e)
               [(constant) (k e)]
               [(identifier) (k (R-lookup e r))]
               [(function)
                (k (lambda (actuals k)
                     (meaning (body-pt e)
                              (extend-env r (formals-pt e) actuals)
                              k)))]
               [(conditional)
                (meaning (test-pt e)
                        r
                        (lambda (v)
                          (if v
                              (meaning (then-pt e) r k)
                              (meaning (else-pt e) r k)))))]
               [(assign)
                (meaning (val-pt e)
                        r
                        (lambda (v)
                          (store-c! (L-lookup (id-pt e) r) v k)))]
               [(sequence) (evaluate-all (exps-pt e) r k)]
               [(call/cc)
                (meaning (fn-pt e)
                        r
                        (lambda (f)
                          (f (list (lambda (actuals kp)
                                     (k (car actuals))))
                              k)))]
               [(application)
                (meaning-of-all (comb-pt e)
                                r
                                (lambda (vals)
                                  ((car vals) (cdr vals) k)))]))]
          [meaning-of-all ...]
          [evaluate-all ...])
    (k meaning)))

```

Figure 25: Haynes, Friedman, and Wand's CPS interpreter for Scheme 84 (part I)

```

(lambda (k)
  (letrec ([meaning
            ;;; Exp * Env * [Val -> Ans] -> Ans
            (lambda (e r k)
              ...)])
    [meaning-of-all
     ;;; List(Exp) * Env * [List(Val) -> Ans] -> Ans
     (lambda (exp-list r k)
       (meaning (car exp-list)
                 r
                 (lambda (val)
                   (if (null? (cdr exp-list))
                       (k (cons val '()))
                       (meaning-of-all (cdr exp-list)
                                       r
                                       (lambda (vals)
                                         (k (cons val
                                                vals)))))))]

     [evaluate-all
      ;;; List(Exp) * Env * [Val -> Ans] -> Ans
      (lambda (exp-list r k)
        (if (null? (cdr exp-list))
            (meaning (car exp-list) r k)
            (meaning (car exp-list)
                      r
                      (lambda (v)
                        (evaluate-all (cdr exp-list) r k))))))]

    (k meaning)))

```

Figure 26: Haynes, Friedman, and Wand's CPS interpreter for Scheme 84 (part II)

- [5] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [16], pages 493–501.
- [6] Ole-Johan Dahl and C.A.R. Hoare. Hierarchical program structures. In Ole-Johan Dahl, Edger Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 157–220. Academic Press, 1972.
- [7] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. Special issue on ESOP'92, the Fourth European Symposium on Programming, Rennes, France, February 1992.

```

(letrec ([meaning
         ;; Exp * Env -> Val
         (lambda (e r)
           (case (type-of-expression e)
             [(constant) e]
             [(identifier) (R-lookup e r)]
             [(function)
              (lambda (actuals)
                (meaning (body-pt e)
                        (extend-env r (formals-pt e) actuals)))]
             [(conditional) (if (meaning (test-pt e) r)
                                (meaning (then-pt e) r)
                                (meaning (else-pt e) r))]
             [(assign)
              (store! (L-lookup (id-pt e) r)
                      (meaning (val-pt e) r))]
             [(sequence) (evaluate-all (exps-pt e) r)]
             [(call/cc)
              (call/cc (lambda (k)
                        ((meaning (fn-pt e) r)
                         (list (lambda (actuals)
                                (throw k (car actuals))))))]
             [(application)
              (let ([vals (meaning-of-all (comb-pt e) r)]
                    ((car vals) (cdr vals)))]))]
         [meaning-of-all
         ;; List(Exp) * Env -> List(Val)
         (lambda (exp-list r)
           (let ([val (meaning (car exp-list) r)])
             (if (null? (cdr exp-list))
                 (cons val '())
                 (cons val (meaning-of-all (cdr exp-list) r)))]))]
         [evaluate-all
         ;; List(Exp) * Env -> Val
         (lambda (exp-list r)
           (if (null? (cdr exp-list))
               (meaning (car exp-list) r)
               (begin
                 (meaning (car exp-list) r)
                 (evaluate-all (cdr exp-list) r)))]))]
  meaning)

```

Figure 27: Direct-style counterpart of Haynes, Friedman, and Wand's interpreter for Scheme 84

- [8] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
- [9] 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.
- [10] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In Hudak [21], pages 157–168.
- [11] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.
- [12] Andrzej Filinski. Representing monads. In Boehm [1], pages 446–457.
- [13] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.
- [14] Michael J. Fischer. Lambda-calculus schemata. In Talcott [43], pages 259–288. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
- [15] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.
- [16] Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.
- [17] Timothy G. Griffin. A formulae-as-types notion of control. In Hudak [21], pages 47–58.
- [18] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [1], pages 458–471.

- [19] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, Texas, August 1984.
- [20] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, Seattle, Washington, March 1990. SIGPLAN Notices, Vol. 25, No. 3.
- [21] Paul Hudak, editor. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 1990. ACM Press.
- [22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
- [23] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In Charles N. Fischer, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 24, No 7, pages 218–226, Portland, Oregon, June 1989. ACM Press.
- [24] Richard A. Kelsey. *Compilation by Program Transformation*. PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, May 1989.
- [25] 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.
- [26] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. Research report, Computer Science Department, Yale University, New Haven, Connecticut, February 1988.
- [27] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.
- [28] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Graham [16], pages 124–136.

- [29] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Rapport de recherche N° 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994. Also appears as Technical report CMU-CS-94-129.
- [30] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
- [31] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.
- [32] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
- [33] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [34] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.
- [35] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.
- [36] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.
- [37] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3/4):233–247, December 1993.
- [38] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 288–298, San Francisco, California, June 1992. ACM Press.

- [39] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [43], pages 289–360.
- [40] 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.
- [41] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [42] Carolyn L. Talcott. *The Essence of $\mathcal{R}um$: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, August 1985.
- [43] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, December 1993.

Recent Publications in the BRICS Report Series

- RS-96-20 Olivier Danvy and Julia L. Lawall. *Back to Direct Style II: First-Class Continuations*. June 1996. 36 pp. A preliminary version of this paper appeared in the proceedings of the 1992 ACM Conference on Lisp and Functional Programming, William Clinger, editor, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.
- RS-96-19 John Hatcliff and Olivier Danvy. *Thunks and the λ -Calculus*. June 1996. 22 pp. To appear in *Journal of Functional Programming*.
- RS-96-18 Thomas Troels Hildebrandt and Vladimiro Sassone. *Comparing Transition Systems with Independence and Asynchronous Transition Systems*. June 1996. 14 pp. To appear in Montanari and Sassone, editors, *Concurrency Theory: 7th International Conference, CONCUR '96 Proceedings*, LNCS 1119, 1996.
- RS-96-17 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. *Eta-Expansion Does The Trick (Revised Version)*. May 1996. 29 pp. To appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*.
- RS-96-16 Lisbeth Fajstrup and Martin Raußen. *Detecting Deadlocks in Concurrent Systems*. May 1996. 10 pp.
- RS-96-15 Olivier Danvy. *Pragmatic Aspects of Type-Directed Partial Evaluation*. May 1996. 27 pp.
- RS-96-14 Olivier Danvy and Karoline Malmkjær. *On the Idempotence of the CPS Transformation*. May 1996. 15 pp.
- RS-96-13 Olivier Danvy and René Vestergaard. *Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*. May 1996. 28 pp. To appear in *8th International Symposium on Programming Languages, Implementations, Logics, and Programs, PLILP '96 Proceedings*, LNCS, 1996.