



ELSEVIER

Science of Computer Programming 22 (1994) 183–195

Science of  
Computer  
Programming

## Back to direct style\*

Olivier Danvy\*\*

*Department of Computer Science, Aarhus University, Ny Munkegade, 8000 Aarhus C, Denmark*

Received January 1993; revised February 1994

---

### Abstract

This paper describes the transformation of  $\lambda$ -terms from continuation-passing style (CPS) to direct style. This transformation is the left inverse of Plotkin's left-to-right call-by-value CPS encoding for the pure  $\lambda$ -calculus.

Not all  $\lambda$ -terms are CPS terms, and not all CPS terms encode a left-to-right call-by-value evaluation. These CPS terms are characterized here; they can be mapped back to direct style. In addition, the two transformations—to continuation-passing style and to direct style—are factored using a language where all intermediate values are named and their computation is sequentialized. The issue of proper tail-recursion is also addressed.

---

Much work has been devoted to transforming programs into continuation-passing style (CPS). (For a recent survey, see Talcott's special issue on continuations [23].) In a CPS program, all procedures take an extra parameter—the continuation—which is a functional accumulator representing “the rest of the computation”. As a consequence, all calls are tail-calls. By contrast, programs that are not in CPS (e.g., programs before CPS transformation) are said to be in “direct style” (DS). Their procedure calls can occur anywhere (i.e., not necessarily in tail position).

---

\* An expanded version is available as Technical Report CIS-92-1, Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA. An earlier version of this paper appeared in: B. Krieg-Brückner, ed., *Proceedings of the 4th European Symposium on Programming, Rennes, France*, Lecture Notes in Computer Science 582 (Springer, Berlin, 1992) 130–150.

This project was initiated during a summer visit to Xerox PARC in 1991 and carried out while the author was working at Kansas State University. This paper was completed during a spring visit to the School of Computer Science of Carnegie Mellon University in 1993, and expurgated while at Aarhus University. Part of this work was supported by NSF under grant CCR-9102625.

\*\*E-mail: danvy@daimi.aau.dk.

In contrast to the work mentioned above, the present paper studies the transformation of CPS programs into DS programs. For brevity, only the pure (call-by-value, with left-to-right evaluation)  $\lambda$ -calculus is considered. A more thorough study (i.e., addressing conditional expressions, primitive operations, block structure, and recursive definitions) is available in a technical report [5].

## 1. From direct style to continuation-passing style and back

The BNF of the pure (direct-style)  $\lambda$ -calculus reads as follows.

$r \in \text{DRoot}$	—DS terms	$r ::= e$
$e \in \text{DExp}$	—DS expressions	$e ::= e_0 e_1 \mid t$
$t \in \text{DTriv}$	—DS trivial expressions	$t ::= x \mid \lambda x.r$
$x \in \text{Ide}$	—identifiers	

Fig. 1 presents a one-pass continuation-passing style (CPS) transformer for the pure  $\lambda$ -calculus with call-by-value evaluation from left to right. This transformer is an optimized version of Plotkin's CPS transformer [19]. It was derived in an earlier work [6] and is only rephrased slightly here to match the syntactic domains. The result of transforming a DS term  $r$  into CPS is given by  $\mathcal{C}[[r]]$ .

These equations can be read as a two-level specification *à la* Nielson and Nielson [17] and thus they can be transliterated in any functional-programming language. Operationally, the overlined  $\lambda$ 's and  $\overline{\text{@}}$ 's correspond to functional

$$\begin{aligned}
 & \mathcal{C} : \text{DRoot} \rightarrow \text{CRoot} \\
 & \mathcal{C}[[e]] = \underline{\lambda}k. \mathcal{C}^{\text{DExp}}[[e]] (\overline{\lambda}t.k \text{@} t) \\
 \\
 & \mathcal{C}^{\text{DExp}} : \text{DExp} \rightarrow [\text{CTriv} \rightarrow \text{CExp}] \rightarrow \text{CExp} \\
 & \mathcal{C}^{\text{DExp}}[[e_0 e_1]] \kappa = \mathcal{C}^{\text{DExp}}[[e_0]] (\overline{\lambda}t_0. \mathcal{C}^{\text{DExp}}[[e_1]] (\overline{\lambda}t_1. (t_0 \text{@} t_1) \text{@} (\underline{\lambda}v. \kappa \overline{\text{@}} v))) \\
 & \mathcal{C}^{\text{DExp}}[[t]] \kappa = \kappa \overline{\text{@}} (\mathcal{C}^{\text{DTriv}}[[t]]) \\
 \\
 & \mathcal{C}^{\text{DTriv}} : \text{DTriv} \rightarrow \text{CTriv} \\
 & \mathcal{C}^{\text{DTriv}}[[x]] = x \\
 & \mathcal{C}^{\text{DTriv}}[[\lambda x.r]] = \underline{\lambda}x. \mathcal{C}[[r]]
 \end{aligned}$$

where  $k$  and the  $v$ 's are fresh variables.

Fig. 1. Call-by-value, left-to-right CPS transformation for the pure  $\lambda$ -calculus.

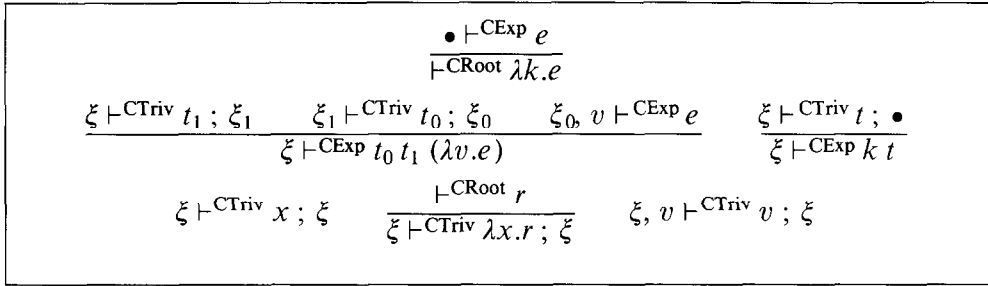


Fig. 2. Occurrence conditions over formal parameters of continuations. The CPS transformation of Fig. 1 performs a particular tree traversal—postfix and left-to-right—yielding a flattened tree [4]. So detecting whether a CPS term encodes a call-by-value and left-to-right evaluation order amounts to parsing a string in reversed-Polish form: with a stack. This is done by scanning the CPS term with a push-down list  $\xi$  holding the formal parameters of continuations.  $\bullet$  denotes the empty list. See Fig. 3 for an example.

abstractions and applications in the translation program (and coincide with the so-called “administrative reductions” [19]), while the underlined  $\lambda$ ’s and  $@$ ’s represent abstract-syntax constructors.<sup>1</sup>

The corresponding BNF of CPS terms reads as follows. (NB: the original identifiers  $x$  coming from the DS term are distinguished from the fresh identifiers  $v$  and  $k$  introduced by  $\mathcal{C}$ . A single identifier  $k$  is sufficient.)

$r \in \text{CRoot}$	—CPS terms	$r ::= \lambda k.e$
$e \in \text{CExp}$	—CPS (serious) expressions	$e ::= t_0 t_1 (\lambda v.e) \mid k t$
$t \in \text{CTriv}$	—CPS trivial expressions	$t ::= x \mid \lambda x.r \mid v$
$x \in \text{Ide}$	—source identifiers	
$v, k \in \text{Var}$	—fresh variables	

The distinction between “serious” and “trivial” expressions is due to Reynolds [20]. Serious terms are passed a continuation—their evaluation may diverge. Trivial terms are passed to a continuation—their evaluation cannot diverge.

A CPS term encodes an evaluation order—here call-by-value—but it also encodes a sequencing order—here from left to right. This sequencing order imposes occurrence conditions over the formal parameters of continuations. The conditions for left-to-right call-by-value are reproduced in Fig. 2. Transforming a DS term  $r$  with  $\mathcal{C}$  yields a CPS term that satisfies the judgement

$$\vdash^{\text{CRoot}} \mathcal{C}[[r]]$$

<sup>1</sup> For example, the two-level expression  $(\lambda x. (\lambda y.y) @ x) @ z$  evaluates to the expression  $(\lambda y.y) z$ .

$$\begin{array}{l}
\mathcal{C}[(f\ x)\ ((g\ x)\ (h\ x))]] \\
= \lambda k.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.h\ x\ (\lambda v_3.v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5)))))) \\
\\
\frac{\bullet, v_5 \vdash^{\text{CTriv}} v_5 ; \bullet}{\bullet, v_5 \vdash^{\text{CExp}} k\ v_5} \\
\frac{\bullet, v_1 \vdash^{\text{CTriv}} v_1 ; \bullet}{\bullet, v_1, v_4 \vdash^{\text{CTriv}} v_4 ; \bullet, v_1} \\
\frac{\bullet, v_1, v_4 \vdash^{\text{CExp}} v_1\ v_4\ (\lambda v_5.k\ v_5)}{\bullet, v_1, v_2 \vdash^{\text{CTriv}} v_2 ; \bullet, v_1} \\
\frac{\bullet, v_1, v_2, v_3 \vdash^{\text{CTriv}} v_3 ; \bullet, v_1, v_2}{\bullet, v_1, v_2, v_3 \vdash^{\text{CExp}} v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5))} \\
\frac{\bullet, v_1, v_2 \vdash^{\text{CTriv}} h ; \bullet, v_1, v_2}{\bullet, v_1, v_2 \vdash^{\text{CExp}} h\ x\ (\lambda v_3.v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5)))} \\
\frac{\bullet, v_1, v_2 \vdash^{\text{CTriv}} x ; \bullet, v_1, v_2}{\bullet, v_1 \vdash^{\text{CTriv}} g ; \bullet, v_1} \\
\frac{\bullet, v_1 \vdash^{\text{CTriv}} x ; \bullet, v_1}{\bullet, v_1 \vdash^{\text{CExp}} g\ x\ (\lambda v_2.h\ x\ (\lambda v_3.v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5)))))} \\
\frac{\bullet \vdash^{\text{CTriv}} f ; \bullet}{\bullet \vdash^{\text{CTriv}} x ; \bullet} \\
\frac{\bullet \vdash^{\text{CExp}} f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.h\ x\ (\lambda v_3.v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5)))))}{\vdash^{\text{CRoot}} \lambda k.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.h\ x\ (\lambda v_3.v_2\ v_3\ (\lambda v_4.v_1\ v_4\ (\lambda v_5.k\ v_5)))))}
\end{array}$$

Fig. 3. Derivation tree for a CPS term.

since the transformation  $\mathcal{C}$  encodes left-to-right call-by-value. Such a CPS term can be mapped back to direct style with the transformation  $\mathcal{D}$  of Fig. 4.<sup>2</sup>

For example, a CPS term such as

$$\lambda k.k\ (\lambda x.\lambda k.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.v_1\ v_2\ (\lambda v_3.k\ v_3))))$$

satisfies the occurrence conditions over formal parameters of continuations

<sup>2</sup>  $\mathcal{D}$  can be derived as follows [5]. Specialize the direct-style denotational semantics of the  $\lambda$ -calculus to CPS terms. Modify the denotation of applications to apply functions to their argument and to an identity continuation, and send the result to the continuation. This suggests an obvious simplification: to apply functions to their argument only instead of to both their argument and the identity continuation. By the same token, since the continuation identifier  $k$  always denotes the identity continuation, its occurrences can be simplified away. The resulting semantics can be taken as a syntax-directed translation  $\mathcal{D}$  of the (CPS)  $\lambda$ -calculus into the (DS)  $\lambda$ -calculus.

$$\begin{aligned}
& \mathcal{D} : \text{CRoot} \rightarrow \text{DRoot} \\
& \mathcal{D}[\lambda k.e] = \mathcal{D}^{\text{CExp}}[e] \rho_{\text{empty}} \\
& \mathcal{D}^{\text{CExp}} : \text{CExp} \rightarrow [\text{Var} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
& \mathcal{D}^{\text{CExp}}[t_0 t_1 (\lambda v.e)] \rho = \mathcal{D}^{\text{CExp}}[e] \rho [v \mapsto (\mathcal{D}^{\text{CTriv}}[t_0] \rho) @ (\mathcal{D}^{\text{CTriv}}[t_1] \rho)] \\
& \mathcal{D}^{\text{CExp}}[k t] \rho = \mathcal{D}^{\text{CTriv}}[t] \rho \\
& \mathcal{D}^{\text{CTriv}} : \text{CTriv} \rightarrow [\text{Var} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
& \mathcal{D}^{\text{CTriv}}[x] \rho = x \\
& \mathcal{D}^{\text{CTriv}}[\lambda x.r] \rho = \lambda x. \mathcal{D}[r] \\
& \mathcal{D}^{\text{CTriv}}[v] \rho = \rho @ v
\end{aligned}$$

Fig. 4. Call-by-value DS transformation for the pure  $\lambda$ -calculus.

$$\begin{array}{c}
\frac{\bullet \vdash^{\text{CExp}} e \triangleright e' \quad \xi \vdash^{\text{CTriv}} t \triangleright t'; \bullet}{\vdash^{\text{CRoot}} \lambda k.e \triangleright e' \quad \xi \vdash^{\text{CExp}} k t \triangleright t'} \\
\frac{\xi \vdash^{\text{CTriv}} t_1 \triangleright t'_1; \xi_1 \quad \xi_1 \vdash^{\text{CTriv}} t_0 \triangleright t'_0; \xi_0 \quad \xi_0, (t'_0 @ t'_1) \vdash^{\text{CExp}} e \triangleright e'}{\xi \vdash^{\text{CExp}} t_0 t_1 (\lambda v.e) \triangleright e'} \\
\frac{\xi \vdash^{\text{CTriv}} x \triangleright x; \xi \quad \vdash^{\text{CRoot}} r \triangleright r' \quad \xi, a \vdash^{\text{CTriv}} v \triangleright a; \xi}{\xi \vdash^{\text{CTriv}} \lambda x.r \triangleright \lambda x.r'; \xi}
\end{array}$$

Fig. 5. Alternative call-by-value DS transformation for the pure  $\lambda$ -calculus.

and thus it can be mapped back to direct style with  $\mathcal{D}$ , yielding

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

Fig. 4 presents  $\mathcal{D}$  equationally to match  $\mathcal{C}$  in Fig. 1. Fig. 5 presents it in logical form to match Fig. 2. A CPS term  $r$  is transformed into a DS term  $r'$  whenever  $r$  satisfies the occurrences conditions over formal parameters of continuations:

$$\frac{\vdash^{\text{CRoot}} r}{\vdash^{\text{CRoot}} r \triangleright r'}$$

The logical presentation makes it more apparent that for satisfactory CPS terms, the substitution carried out with an environment  $\rho$  in Fig. 4 can actually be carried out with a stack  $\xi$  in Fig. 5.

But most CPS terms break the stack discipline of Fig. 2. For example, a term such as

$$\lambda k.k (\lambda x.\lambda k.g x (\lambda v_2.f x (\lambda v_1.v_1 v_2 (\lambda v_3.k v_3))))$$

does not satisfy the occurrence conditions over formal parameters of continuations because  $(g x)$  is computed before  $(f x)$  but its result  $v_2$  is used after the result  $v_1$  of  $(f x)$ .

This unsatisfactory CPS term, however, can be rewritten as another one satisfying the occurrence conditions of Fig. 2:

$$\lambda k.k (\lambda x.\lambda k.g x (\lambda v_2.[\lambda v.\lambda k.f x (\lambda v_1.v_1 v (\lambda v_4.k v_4))] v_2 (\lambda v_3.k v_3)))$$

This rewritten CPS term can be mapped back to direct style with  $\mathcal{D}$ , yielding

$$\lambda x.[\lambda v.(f x) v] (g x)$$

where we have used brackets instead of parentheses to improve readability. The rewriting is simply an  $\eta$ -expansion in CPS, ensuring that  $(g x)$  is evaluated before  $(f x)$ .

## 2. Staging the CPS and the DS transformations

The effect of the CPS transformation can be separated into stages [4,14,24]. Starting with a DS term, all intermediate values can be named with a let form, their computation can be sequentialized, and all trivial terms can be coerced into serious ones with a return form. Then continuations can be introduced

$$\begin{aligned} \mathcal{E}_d &: \text{DRoot} \rightarrow \text{IRoot} \\ \mathcal{E}_d[e] &= \mathcal{E}_d^{\text{DExp}}[e] (\bar{\lambda}t.\text{return}(t)) \\ \\ \mathcal{E}_d^{\text{DExp}} &: \text{DExp} \rightarrow [\text{ITriv} \rightarrow \text{IExp}] \rightarrow \text{IExp} \\ \mathcal{E}_d^{\text{DExp}}[e_0 e_1] \kappa &= \mathcal{E}_d^{\text{DExp}}[e_0] (\bar{\lambda}t_0.\mathcal{E}_d^{\text{DExp}}[e_1] (\bar{\lambda}t_1.\text{let } v = t_0 \text{ @ } t_1 \text{ in } \kappa \text{ @ } v)) \\ \mathcal{E}_d^{\text{DExp}}[t] \kappa &= \kappa \text{ @ } (\mathcal{E}_d^{\text{DTriv}}[t]) \\ \\ \mathcal{E}_d^{\text{DTriv}} &: \text{DTriv} \rightarrow \text{ITriv} \\ \mathcal{E}_d^{\text{DTriv}}[x] &= x \\ \mathcal{E}_d^{\text{DTriv}}[\lambda x.r] &= \lambda x.\mathcal{E}_d[r] \end{aligned}$$

where the  $v$ 's are fresh variables.

Fig. 6. DS encoding into a  $\lambda$ -language with return and let.

$$\begin{aligned}
& \mathcal{E}_c : \text{CRoot} \rightarrow \text{IRoot} \\
& \mathcal{E}_c[\lambda k.e] = \mathcal{E}_c^{\text{CExp}}[e] \\
& \mathcal{E}_c^{\text{CExp}} : \text{CExp} \rightarrow \text{IExp} \\
& \mathcal{E}_c^{\text{CExp}}[t_0 t_1 (\lambda v.e)] = \underline{\text{let}} v = (\mathcal{E}_c^{\text{CTriv}}[t_0]) \underline{\text{@}} (\mathcal{E}_c^{\text{CTriv}}[t_1]) \underline{\text{in}} \mathcal{E}_c^{\text{CExp}}[e] \\
& \mathcal{E}_c^{\text{CExp}}[\lambda k t] = \underline{\text{return}} (\mathcal{E}_c^{\text{CTriv}}[t]) \\
& \mathcal{E}_c^{\text{CTriv}} : \text{CTriv} \rightarrow \text{IExp} \\
& \mathcal{E}_c^{\text{CTriv}}[x] = x \\
& \mathcal{E}_c^{\text{CTriv}}[\lambda x.r] = \underline{\lambda} x. \mathcal{E}_c[r] \\
& \mathcal{E}_c^{\text{CTriv}}[v] = v
\end{aligned}$$

Fig. 7. CPS encoding into a  $\lambda$ -language with return and let (continuation elimination).

without any further syntax shuffling. Fig. 6 presents the encoding of  $\lambda$ -terms into an intermediate  $\lambda$ -language containing return and let. The corresponding BNF of intermediate terms reads as follows.

$$\begin{aligned}
r \in \text{IRoot} & \quad \text{—intermediate terms} & r ::= e \\
e \in \text{IExp} & \quad \text{—intermediate (serious) expressions} & e ::= \text{let } v = t_0 t_1 \text{ in } e \mid \text{return}(t) \\
t \in \text{ITriv} & \quad \text{—intermediate trivial expressions} & t ::= x \mid \lambda x.r \mid v \\
x \in \text{Ide} & \quad \text{—source identifiers} \\
v \in \text{Var} & \quad \text{—fresh variables}
\end{aligned}$$

Unfolding the form

$$\text{let } v = s \text{ in } e$$

as

$$(\bar{\lambda} v.e) \bar{\text{@}} s$$

i.e., by substituting  $s$  for  $v$  in  $e$  (which is safe because the let parameters occur similarly as the continuation parameters in Fig. 2) undoes the encoding of Fig. 6 and thus yields a DS term (see Fig. 8 for details<sup>3</sup>). Translating the let form as

<sup>3</sup> Again, Fig. 8 could be expressed in logical form as Figs 2 and 5, and the substitution could be carried out with a stack, as in Fig. 5.

$$\begin{aligned}
\mathcal{B} &: \text{IRoot} \rightarrow \text{DRoot} \\
\mathcal{B}[e] &= \mathcal{B}^{\text{IExp}}[e] \rho_{\text{empty}} \\
\\
\mathcal{B}^{\text{IExp}} &: \text{IExp} \rightarrow [\text{Var} \rightarrow \text{DExp}] \rightarrow \text{DExp} \\
\mathcal{B}^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] \rho &= \mathcal{B}^{\text{IExp}}[e] \rho [v \mapsto (\mathcal{B}^{\text{ITriv}}[t_0] \rho) \ @ \ (\mathcal{B}^{\text{ITriv}}[t_1] \rho)] \\
\mathcal{B}^{\text{IExp}}[\text{return}(t)] \rho &= \mathcal{B}^{\text{ITriv}}[t] \rho \\
\\
\mathcal{B}^{\text{ITriv}} &: \text{ITriv} \rightarrow \text{DTriv} \\
\mathcal{B}^{\text{ITriv}}[x] \rho &= x \\
\mathcal{B}^{\text{ITriv}}[\lambda x. r] \rho &= \lambda x. \mathcal{B}[r] \\
\mathcal{B}^{\text{ITriv}}[v] \rho &= \rho \ @ \ v
\end{aligned}$$

Fig. 8. Back to direct style.

$$\begin{aligned}
\mathcal{F} &: \text{IRoot} \rightarrow \text{CRoot} \\
\mathcal{F}[e] &= \lambda k. \mathcal{F}^{\text{IExp}}[e] k \\
\\
\mathcal{F}^{\text{IExp}} &: \text{IExp} \rightarrow \text{Var} \rightarrow \text{CExp} \\
\mathcal{F}^{\text{IExp}}[\text{let } v = t_0 \ t_1 \text{ in } e] k &= (\mathcal{F}^{\text{ITriv}}[t_0] \ @ \ \mathcal{F}^{\text{ITriv}}[t_1]) \ @ \ (\lambda v. \mathcal{F}^{\text{IExp}}[e] k) \\
\mathcal{F}^{\text{IExp}}[\text{return}(t)] k &= k \ @ \ (\mathcal{F}^{\text{ITriv}}[t]) \\
\\
\mathcal{F}^{\text{ITriv}} &: \text{ITriv} \rightarrow \text{CTriv} \\
\mathcal{F}^{\text{ITriv}}[x] &= x \\
\mathcal{F}^{\text{ITriv}}[\lambda x. r] &= \lambda x. \mathcal{F}[r] \\
\mathcal{F}^{\text{ITriv}}[v] &= v
\end{aligned}$$

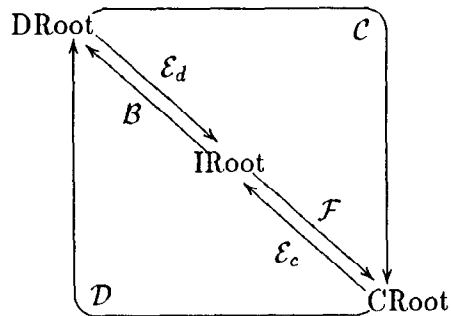
where  $k$  is a fresh variable.

Fig. 9. Forth to continuation-passing style (continuation introduction).

$$s \ @ \ (\lambda v. e)$$

amounts to introducing continuations and thus yields a CPS term (see Fig. 9 for details).

The situation is summarized in the following diagram.



Going back to the example of Section 1,  $\mathcal{E}_d$  transforms the DS term

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

into the intermediate-style term

$$\begin{aligned} &\lambda x.\text{let } v_1 = f x \\ &\quad \text{in let } v_2 = g x \\ &\quad \quad \text{in let } v_3 = v_1 v_2 \\ &\quad \quad \quad \text{in return}(v_3) \end{aligned}$$

which  $\mathcal{F}$  transforms into the CPS term

$$\lambda k.k (\lambda x.\lambda k.f x (\lambda v_1.g x (\lambda v_2.v_1 v_2 (\lambda v_3.k v_3))))).$$

$\mathcal{E}_c$  maps this CPS term into the intermediate-style term above, and  $\mathcal{B}$  maps this intermediate-style term back to the DS term above.

The mapping between CPS terms and intermediate terms is a bijection [12]. The intermediate terms coincide with Moggi's "monadic" language,  $\mathcal{B}$  in Fig. 8 corresponds to the identity monad, and  $\mathcal{F}$  in Fig. 9 corresponds to the continuation monad [9,11,16]. Finally, coercing a CPS term into another one that satisfies the occurrence conditions over the parameters of continuations, i.e.,  $\eta$ -expansion, corresponds to using a strict binding construct in the DS  $\lambda$ -calculus.

### 3. Proper tail recursion

Sometimes it is important to process tail-calls properly, e.g., in the implementation of a programming language such as Scheme [2]. The transformations  $\mathcal{C}$  and  $\mathcal{E}_d$  are not properly tail-recursive encodings. This is easily seen in the common example of Sections 1 and 2:

In the body of the  $\lambda$ -abstraction

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

the outer call is a tail-call—i.e., the result of this call is also the result of the call to the  $\lambda$ -abstraction. However, in the intermediate encoding,

$$\begin{aligned} &\lambda x.\text{let } v_1 = f\ x \\ &\quad \text{in let } v_2 = g\ x \\ &\quad \quad \text{in let } v_3 = v_1\ v_2 \\ &\quad \quad \quad \text{in return}(v_3) \end{aligned}$$

the call  $v_1\ v_2$  is not a tail-call anymore. A properly tail-recursive encoding would be the following one.

$$\begin{aligned} &\lambda x.\text{let } v_1 = f\ x \\ &\quad \text{in let } v_2 = g\ x \\ &\quad \quad \text{in } v_1\ v_2 \end{aligned}$$

Similarly, in the CPS encoding,

$$\lambda k.k\ (\lambda x.\lambda k.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.v_1\ v_2\ (\lambda v_3.k\ v_3))))$$

the continuation of  $v_1\ v_2$  intensionally is not the same as  $k$ . A properly tail-recursive encoding would be the following one.

$$\lambda k.k\ (\lambda x.\lambda k.f\ x\ (\lambda v_1.g\ x\ (\lambda v_2.v_1\ v_2\ k)))$$

Such an encoding can make a significant difference in a CPS compiler such as the one for Standard ML of New Jersey [1] (Trevor Jim and Andrew Appel, personal communication, San Francisco, California, June 1992).

Here are the BNFs of intermediate terms and of CPS terms that enable a properly tail-recursive encoding:

$$\begin{array}{ll} r \in \text{IRoot} & r ::= e \\ e \in \text{IExp} & e ::= \text{let } v = t_0\ t_1 \text{ in } e \mid t_0\ t_1 \mid \text{return}(t) \\ t \in \text{ITriv} & t ::= x \mid \lambda x.r \mid v \\ x \in \text{Ide} & \\ v \in \text{Var} & \\ \\ r \in \text{CRoot} & r ::= \lambda k.e \\ e \in \text{CExp} & e ::= t_0\ t_1\ (\lambda v.e) \mid t_0\ t_1\ k \mid k\ t \\ t \in \text{CTriv} & t ::= x \mid \lambda x.r \mid v \\ x \in \text{Ide} & \\ v, k \in \text{Var} & \end{array}$$

The diligent reader is left with the exercise of updating the figures to match these two BNFs. (Hint: see [6, Fig. 3].)

#### 4. Related work

Sabry and Felleisen's recent work on equational reasoning about CPS programs [21] aims at understanding precisely what in CPS transformation enables, e.g., compile-time optimizations. This leads them to define an "un-CPSer" that only partly corresponds to the DS transformer for the pure  $\lambda$ -calculus in that:

- (1) It does not consider the occurrence conditions over the formal parameters of continuations (see Fig. 2). This is because the un-CPSer is not used as a source-to-source program transformer but rather aims at relating terms that have the same meaning.
- (2) It performs the steps of Fig. 7 but represents let expressions as  $\beta$ -redexes. Also, it does not use return expressions.
- (3) It does not perform the administrative reductions corresponding to unfolding the let expressions (and enabled by the occurrence conditions over the formal parameters of continuations).

Sabry and Felleisen too notice and rely on the uniqueness of a continuation identifier  $k$ . Their CPS transformer also performs additional reductions for DS  $\beta$ -redexes.

In "Back to direct style II" [7], the unicity of  $k$  is relaxed by letting any lexically visible continuation identifier be applied, instead of only the current one. In the DS world, this amounts to introducing control operators such as **call/cc** to declare a first-class continuation and **throw** to send a value to a first-class continuation. More generally, relaxing the CPS texture to allow non-tail calls would amount to introducing control operators and delimiters such as **shift** and **reset** [6,8,9,13].

Other stagings of the CPS transformation are possible. For example, it is possible to (1) name intermediate values, (2) introduce named continuations, and (3) inline these continuations, obtaining a CPS term. In fact, it is even possible to factor out sequentialization from the CPS transformation by creating a new step (2'). This new step amounts to deciding in which order the intermediate continuations should be inlined. Each of these steps is reversible. This staging is described elsewhere [14].

Intermediate languages such as the one of Section 2 are currently being investigated as an alternative to CPS. Flanagan et al. address the problem of compiling with and without continuations [10]. Finally, to have or not to have continuations can influence the precision of flow analyses, as studied elsewhere [3,22].

## Acknowledgements

Karoline Malmkjær kindly confirmed the BNF of CPS terms using abstract interpretation [15]. Frank Pfenning demonstrated his Elf system [18] by checking the contents of the figures. I am also grateful to Andrzej Filinski, John Hatcliff, and Julia Lawall for their interaction. The diagram of Section 2 was drawn with Kristoffer Rose's XY-pic package.

## References

- [1] A.W. Appel, *Compiling with Continuations* (Cambridge University Press, Cambridge, England, 1992).
- [2] W. Clinger and J. Rees (eds.), *Revised<sup>4</sup> report on the algorithmic language Scheme*, LISP Pointers IV (3) (ACM Press, New York, 1991) 1–55.
- [3] C. Consel and O. Danvy, For a better support of static data flow, in: J. Hughes, ed., *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture, Cambridge, MA*, Lecture Notes in Computer Science 523 (Springer, Berlin, 1991) 496–519.
- [4] O. Danvy, Three steps for the CPS transformation, Tech. Report CIS-92-2, Kansas State University, Manhattan, KS (1991).
- [5] O. Danvy, Back to direct style (revised and extended version), Tech. Report CIS-92-1, Kansas State University, Manhattan, KS (1993).
- [6] O. Danvy and A. Filinski, Representing control, a study of the CPS transformation, *Math. Structures Comput. Sci.* 2 (4) (1992) 361–391.
- [7] O. Danvy and J.L. Lawall, Back to direct style II: first-class continuations, in: W. Clinger, ed., *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, San Francisco, CA*, LISP Pointers V (1) (ACM Press, New York, 1992) 299–310.
- [8] M. Felleisen, The theory and practice of first-class prompts, in: J. Ferrante and P. Mager, eds., *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA* (ACM Press, New York, 1988) 180–190.
- [9] A. Filinski, Representing monads, in: H.-J. Boehm, ed., *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, OR* (ACM Press, New York, 1994) 446–457.
- [10] C. Flanagan, A. Sabry, B.F. Duba and M. Felleisen, The essence of compiling with continuations, in: D.W. Wall, ed., *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation, Albuquerque, NM, SIGPLAN Not.* 28 (6) (1993) 237–247.
- [11] J. Hatcliff and O. Danvy, A generic account of continuation-passing styles, in: H.-J. Boehm, ed., *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, OR* (ACM Press, New York, 1994) 458–471.
- [12] J.L. Lawall, Proofs by structural induction using partial evaluation, in: D.A. Schmidt, ed., *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark* (ACM Press, New York, 1993) 155–166.
- [13] J.L. Lawall, Ph.D. Thesis, Computer Science Department, Indiana University, Bloomington, IN (to appear).
- [14] J.L. Lawall and O. Danvy, Separating stages in the continuation-passing style transformation, in: S.L. Graham, ed., *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, SC* (ACM Press, New York, 1993) 124–136.
- [15] K. Malmkjær, Abstract interpretation of partial-evaluation algorithms, Ph.D. Thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, KS (1993),

- [16] E. Moggi, Notions of computation and monads, *Inform. Comput.* **93** (1991) 55–92.
- [17] F. Nielson and H.R. Nielson, Two-level semantics and code generation, *Theoret. Comput. Sci.* **56** (1) (1988) 59–133; Special issue on ESOP'86, the First European Symposium on Programming, Saarbrücken, March 17–19, 1986.
- [18] F. Pfenning, Logic programming in the LF logical framework, in: G. Huet and G.D. Plotkin, eds., *Logical Frameworks* (Cambridge University Press, Cambridge, England, 1991) 149–181.
- [19] G.D. Plotkin, Call-by-name, call-by-value and the  $\lambda$ -calculus, *Theoret. Comput. Sci.* **1** (1975) 125–159.
- [20] J.C. Reynolds, Definitional interpreters for higher-order programming languages, in: *Proceedings of the 25th ACM National Conference*, Boston, MA (1972) 717–740.
- [21] A. Sabry and M. Felleisen, Reasoning about programs in continuation-passing style, in: W. Clinger, ed., *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, CA, LISP Pointers V (1) (ACM Press, New York, 1992) 288–298.
- [22] A. Sabry and M. Felleisen, Is continuation-passing useful for data flow analysis? in: V. Sarkar, ed., *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, Orlando, FL (ACM Press, New York, to appear).
- [23] C.L. Talcott, ed., *LISP Symbolic Comput.* **6** (3/4), **7** (1) (1993–1994); Special Issue on Continuations.
- [24] D. Weise, Advanced compiling techniques, Course Notes at Stanford University, Stanford, CA (1990).