Back to Direct Style [*]

Olivier Danvy [**]
Department of Computer Science
Aarhus University [***]
(danvy@daimi.aau.dk)

February 22, 1994

Abstract

This paper describes the transformation of λ-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 λ-calculus.

Not all λ-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 [25].) 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).

In contrast to the work mentioned above, the present paper studies the transformation of CPS programs into DS. For brevity, only the pure (call-by-value, with left-to-right evaluation) λ-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 [7].

[*] *Science of Computer Programming*, Vol. 22, pp. 183–195. A expanded version is available as Technical report CIS-92-1, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas 66506, USA. An earlier version of this paper appeared in the Proceedings of the 4th European Symposium on Programming, Bernd Krieg-Brückner, editor, number 582 in Lecture Notes in Computer Science, pages 130–150, Rennes, France, February 1992.

[**] 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. This work was partly supported by NSF under grant CCR-9102625 and by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

[***] Ny Munkegade, 8000 Aarhus C, Denmark.

1 From direct style to continuation-passing style and back

The BNF of the pure (direct-style) λ-calculus reads as follows.

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

Figure 1 presents a one-pass continuation-passing style (CPS) transformer for the pure λ-calculus with call-by-value evaluation from left to right. This transformer is an optimized version of Plotkin's CPS transformer [21]. It was derived in an earlier work [8] 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 [19] and thus they can be transliterated in any functional-programming language. Operationally, the overlined $\overline{\lambda}$'s and $\overline{@}$'s correspond to functional abstractions and applications in the translation program (and coincide with the so-called "administrative reductions" [21]), while the underlined $\underline{\lambda}$'s and $\underline{@}$'s represent abstract-syntax constructors.[1]

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.)

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

The distinction between "serious" and "trivial" expressions is due to Reynolds [22]. 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 Figure 2. Transforming a DS term r with $\mathcal{C}$ yields a CPS term that satisfies the judgement

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

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 Figure 4.[2]

[1]For example, the two-level expression $(\overline{\lambda} x.(\underline{\lambda} y.y)\,\underline{@}\,x)\,\overline{@}\,z$ evaluates to the expression $(\lambda y.y)\, z$.

[2]$\mathcal{D}$ can be derived as follows [7]. Specialize the direct-style denotational semantics of the λ-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

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

where k and the v's are fresh variables.

Figure 1: Call-by-value, left-to-right CPS transformation for the pure λ-calculus

$$\frac{\bullet \vdash^{\text{CExp}} e}{\vdash^{\text{CRoot}} \lambda k.e}$$

$$\frac{\xi \vdash^{\text{CTriv}} t_1 \;;\; \xi_1 \qquad \xi_1 \vdash^{\text{CTriv}} t_0 \;;\; \xi_0 \qquad \xi_0,\, v \vdash^{\text{CExp}} e}{\xi \vdash^{\text{CExp}} t_0\,t_1\,(\lambda v.e)} \qquad\qquad \frac{\xi \vdash^{\text{CTriv}} t \;;\; \bullet}{\xi \vdash^{\text{CExp}} k\,t}$$

$$\xi \vdash^{\text{CTriv}} x \;;\; \xi \qquad\qquad \frac{\vdash^{\text{CRoot}} r}{\xi \vdash^{\text{CTriv}} \lambda x.r \;;\; \xi} \qquad\qquad \xi,\, v \vdash^{\text{CTriv}} v \;;\; \xi$$

Figure 2: Occurrence conditions over formal parameters of continuations

The CPS transformation of Figure 1 performs a particular tree traversal — postfix and left-to-right — yielding a flattened tree [6]. 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 ξ holding the formal parameters of continuations. $\bullet$ denotes the empty list. See Figure 3 for an example.

$$
\begin{array}{ll}
& \mathcal{C}\llbracket (f\,x)\,((g\,x)\,(h\,x)) \rrbracket \\
= & \lambda k.f\,x\,(\lambda v_1.g\,x\,(\lambda v_2.h\,x\,(\lambda v_3.v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5)))))
\end{array}
$$

$$
\dfrac{\dfrac{\begin{array}{c}
\dfrac{\begin{array}{c}
\dfrac{\begin{array}{c}
\dfrac{\begin{array}{c}
\dfrac{\begin{array}{c}
\dfrac{\bullet,\, v_5 \vdash^{\text{CTriv}} v_5 \;;\; \bullet}{\bullet,\, v_5 \vdash^{\text{CExp}} k\,v_5} \\
\bullet,\, v_1 \vdash^{\text{CTriv}} v_1 \;;\; \bullet \\
\bullet,\, v_1,\, v_4 \vdash^{\text{CTriv}} v_4 \;;\; \bullet,\, v_1
\end{array}}{\bullet,\, v_1,\, v_4 \vdash^{\text{CExp}} v_1\,v_4\,(\lambda v_5.k\,v_5)} \\
\bullet,\, v_1,\, v_2 \vdash^{\text{CTriv}} v_2 \;;\; \bullet,\, v_1 \\
\bullet,\, v_1,\, v_2,\, v_3 \vdash^{\text{CTriv}} v_3 \;;\; \bullet,\, v_1,\, v_2
\end{array}}{\bullet,\, v_1,\, v_2,\, v_3 \vdash^{\text{CExp}} v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5))} \\
\bullet,\, v_1,\, v_2 \vdash^{\text{CTriv}} h \;;\; \bullet,\, v_1,\, v_2 \\
\bullet,\, v_1,\, v_2 \vdash^{\text{CTriv}} x \;;\; \bullet,\, v_1,\, v_2
\end{array}}{\bullet,\, v_1,\, v_2 \vdash^{\text{CExp}} h\,x\,(\lambda v_3.v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5)))} \\
\bullet,\, v_1 \vdash^{\text{CTriv}} g \;;\; \bullet,\, v_1 \\
\bullet,\, v_1 \vdash^{\text{CTriv}} x \;;\; \bullet,\, v_1
\end{array}}{\bullet,\, v_1 \vdash^{\text{CExp}} g\,x\,(\lambda v_2.h\,x\,(\lambda v_3.v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5))))} \\
\bullet \vdash^{\text{CTriv}} f \;;\; \bullet \\
\bullet \vdash^{\text{CTriv}} x \;;\; \bullet
\end{array}}{\bullet \vdash^{\text{CExp}} f\,x\,(\lambda v_1.g\,x\,(\lambda v_2.h\,x\,(\lambda v_3.v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5)))))}}{\vdash^{\text{CRoot}} \lambda k.f\,x\,(\lambda v_1.g\,x\,(\lambda v_2.h\,x\,(\lambda v_3.v_2\,v_3\,(\lambda v_4.v_1\,v_4\,(\lambda v_5.k\,v_5)))))}
$$

Figure 3: Derivation tree for a CPS term

$$
\begin{array}{rcl}
\mathcal{D} & : & \text{CRoot} \to \text{DRoot} \\
\mathcal{D}\llbracket \lambda k.e \rrbracket & = & \mathcal{D}^{\text{CExp}}\llbracket e \rrbracket\,\rho_{empty} \\
\\
\mathcal{D}^{\text{CExp}} & : & \text{CExp} \to [\text{Var} \to \text{DExp}] \to \text{DExp} \\
\mathcal{D}^{\text{CExp}}\llbracket t_0\,t_1\,(\lambda v.e) \rrbracket\,\rho & = & \mathcal{D}^{\text{CExp}}\llbracket e \rrbracket\,\rho[v \,\overline{\mapsto}\, (\mathcal{D}^{\text{CTriv}}\llbracket t_0 \rrbracket\,\rho)\,\underline{@}\,(\mathcal{D}^{\text{CTriv}}\llbracket t_1 \rrbracket\,\rho)] \\
\mathcal{D}^{\text{CExp}}\llbracket k\,t \rrbracket\,\rho & = & \mathcal{D}^{\text{CTriv}}\llbracket t \rrbracket\,\rho \\
\\
\mathcal{D}^{\text{CTriv}} & : & \text{CTriv} \to [\text{Var} \to \text{DExp}] \to \text{DExp} \\
\mathcal{D}^{\text{CTriv}}\llbracket x \rrbracket\,\rho & = & x \\
\mathcal{D}^{\text{CTriv}}\llbracket \lambda x.r \rrbracket\,\rho & = & \underline{\lambda} x.\mathcal{D}\llbracket r \rrbracket \\
\mathcal{D}^{\text{CTriv}}\llbracket v \rrbracket\,\rho & = & \rho\,\overline{@}\,v
\end{array}
$$

Figure 4: Call-by-value DS transformation for the pure λ-calculus

$$\frac{\bullet \vdash^{\text{CExp}} e \,\rhd\, e'}{\vdash^{\text{CRoot}} \lambda k.e \,\rhd\, e'} \qquad \frac{\xi \vdash^{\text{CTriv}} t \,\rhd\, t' \,;\, \bullet}{\xi \vdash^{\text{CExp}} k\,t \,\rhd\, t'}$$

$$\frac{\xi \vdash^{\text{CTriv}} t_1 \,\rhd\, t_1' \,;\, \xi_1 \qquad \xi_1 \vdash^{\text{CTriv}} t_0 \,\rhd\, t_0' \,;\, \xi_0 \qquad \xi_0, (t_0' \,\underline{@}\, t_1') \vdash^{\text{CExp}} e \,\rhd\, e'}{\xi \vdash^{\text{CExp}} t_0\, t_1\, (\lambda v.e) \,\rhd\, e'}$$

$$\xi \vdash^{\text{CTriv}} x \,\rhd\, x \,;\, \xi \qquad \frac{\vdash^{\text{CRoot}} r \,\rhd\, r'}{\xi \vdash^{\text{CTriv}} \lambda x.r \,\rhd\, \lambda x.r' \,;\, \xi} \qquad \xi, a \vdash^{\text{CTriv}} v \,\rhd\, a \,;\, \xi$$

Figure 5: Alternative call-by-value DS transformation for the pure λ-calculus

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

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

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

$$\frac{\vdash^{\text{CRoot}} r}{\vdash^{\text{CRoot}} r \,\rhd\, r'}$$

The logical presentation makes it more apparent that for satisfactory CPS terms, the substitution carried out with an environment ρ in Figure 4 can actually be carried out with a stack ξ in Figure 5.

But most CPS terms break the stack discipline of Figure 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 Figure 2:

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

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) λ-calculus into the (DS) λ-calculus.

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 η-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 [6, 16, 26]. 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 without any further syntax shuffling. Figure 6 presents the encoding of λ-terms into an intermediate λ-language containing return and let. The corresponding BNF of intermediate terms reads as follows.

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

Unfolding the form

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

as

$$(\overline{\lambda} v.e)\,\overline{@}\,s$$

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

$$s\,\underline{@}\,(\underline{\lambda} v.e)$$

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

The situation is summarized in the following diagram.

$$\begin{tikzcd}
\text{DRoot} \arrow[dr, shift left, "\mathcal{E}_d"] \arrow[ddrr, bend left=60, "\mathcal{C}"] & & \\
& \text{IRoot} \arrow[ul, shift left, "\mathcal{B}"] \arrow[dr, shift left, "\mathcal{F}"] & \\
& & \text{CRoot} \arrow[ul, shift left, "\mathcal{E}_c"] \arrow[uull, bend left=60, "\mathcal{D}"]
\end{tikzcd}$$

[3] Again, Figure 8 could be expressed in logical form as Figures 2 and 5, and the substitution could be carried out with a stack, as in Figure 5.

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

where the v's are fresh variables.

Figure 6: DS encoding into a λ-language with return and let

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

Figure 7: CPS encoding into a λ-language with return and let (continuation elimination)

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

Figure 8: Back to direct style

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

where k is a fresh variable.

Figure 9: Forth to continuation-passing style (continuation introduction)

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

$$\begin{array}{l}\lambda x.\,\mathrm{let}\ v_1 = f\ x\\ \quad\ \mathrm{in\ let}\ v_2 = g\ x\\ \qquad\ \mathrm{in\ let}\ v_3 = v_1\ v_2\\ \qquad\quad\ \mathrm{in\ return}(v_3)\end{array}$$

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 [14]. The intermediate terms coincide with Moggi's "monadic" language, $\mathcal{B}$ in Figure 8 corresponds to the identity monad, and $\mathcal{F}$ in Figure 9 corresponds to the continuation monad [11, 13, 18]. Finally, coercing a CPS term into another one that satisfies the occurrence conditions over the parameters of continuations, i.e., η-expansion, corresponds to using a strict binding construct in the DS λ-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 [4]. 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 λ-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 λ-abstraction. However, in the intermediate encoding,

$$\begin{array}{l}\lambda x.\,\mathrm{let}\ v_1 = f\ x\\ \quad\ \mathrm{in\ let}\ v_2 = g\ x\\ \qquad\ \mathrm{in\ let}\ v_3 = v_1\ v_2\\ \qquad\quad\ \mathrm{in\ return}(v_3)\end{array}$$

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

$$\begin{array}{l}\lambda x.\,\mathrm{let}\ v_1 = f\ x\\ \quad\ \mathrm{in\ let}\ v_2 = g\ x\\ \qquad\ \mathrm{in}\ v_1\ v_2\end{array}$$

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:

$$\begin{array}{rll}
r & \in \mathrm{IRoot} & r ::= e \\
e & \in \mathrm{IExp} & e ::= \mathrm{let}\ v = t_0\,t_1\ \mathrm{in}\ e \mid t_0\,t_1 \mid \mathrm{return}(t) \\
t & \in \mathrm{ITriv} & t ::= x \mid \lambda x.r \mid v \\
x & \in \mathrm{Ide} & \\
v & \in \mathrm{Var} &
\end{array}$$

$$\begin{array}{rll}
r & \in \mathrm{CRoot} & r ::= \lambda k.e \\
e & \in \mathrm{CExp} & e ::= t_0\,t_1\,(\lambda v.e) \mid t_0\,t_1\,k \mid k\,t \\
t & \in \mathrm{CTriv} & t ::= x \mid \lambda x.r \mid v \\
x & \in \mathrm{Ide} & \\
v, k & \in \mathrm{Var} &
\end{array}$$

The diligent reader is left with the exercise of updating the figures to match these two BNFs. (Hint: see [8, Fig. 3].)

4 Related work

Sabry and Felleisen's recent work on equational reasoning about CPS programs [23] 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 λ-calculus in that:

1. It does not consider the occurrence conditions over the formal parameters of continuations (see Figure 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 Figure 7 but represents let expressions as β-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 β-redexes.

In "Back to Direct Style II" [9], 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** [8, 10, 11, 15].

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 [16].

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 [12]. Finally, to have or not to have continuations can influence the precision of flow analyses, as studied elsewhere [5, 24].

Acknowledgements

Karoline Malmkjær kindly confirmed the BNF of CPS terms using abstract interpretation [17]. Frank Pfenning demonstrated his Elf system [20] 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] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[3] William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[4] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991.

[6] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991.

[7] Olivier Danvy. Back to direct style (revised and extended version). Technical Report CIS-92-1, Kansas State University, Manhattan, Kansas, 1993.

[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 Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [3], pages 299–310.

[10] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.

[11] Andrzej Filinski. Representing monads. In Boehm [2], pages 446–457.

[12] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[13] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [2], pages 458–471.

[14] Julia L. Lawall. Proofs by structural induction using partial evaluation. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 155–166, Copenhagen, Denmark, June 1993. ACM Press.

[15] Julia L. Lawall. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, 1994. Forthcoming.

[16] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina, January 1993. ACM Press.

[17] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms.* PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, USA, March 1993.

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

[19] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988. Special issue on ESOP'86, the First European Symposium on Programming, Saarbrücken, March 17-19, 1986.

[20] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

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

[22] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.

[23] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [3], pages 288–298.

[24] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices (to appear), Orlando, Florida, June 1994. ACM Press.

[25] Carolyn L. Talcott, editor. *Special issue on continuations*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4 and Vol. 7, No. 1. Kluwer Academic Publishers, 1993-1994.

[26] Daniel Weise. Advanced compiling techniques. Course notes at Stanford University, 1990.