The Occurrence of Continuation Parameters in CPS Terms

Olivier Danvy   Frank Pfenning

February 1995
CMU-CS-95-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We prove an occurrence property about formal parameters of continuations in Continuation-Passing Style (CPS) terms that have been automatically produced by CPS transformation of pure, call-by-value λ-terms. Essentially, parameters of continuations obey a stack-like discipline.
This property was introduced, but not formally proven, in an earlier work on the Direct-Style transformation (the inverse of the CPS transformation). The proof has been implemented in Elf, a constraint logic programming language based on the logical framework LF. In fact, it was the implementation that inspired the proof. Thus this note also presents a case study of machine-assisted proof discovery.
All the programs are available in

$$\left\{\begin{array}{l}\texttt{ftp.daimi.aau.dk:pub/danvy/Programs/danvy-pfenning-Elf93.tar.gz}\\ \texttt{ftp.cs.cmu.edu:user/fp/papers/cpsocc95.tar.gz}\end{array}\right.$$

Most of the research reported here was carried out while the first author visited Carnegie Mellon University in the Spring of 1993. Current address: Olivier Danvy, Ny Munkegade, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark; danvy@daimi.aau.dk

This work was supported by NSF Grant CCR-93-03383 and by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

Keywords: Lambda-Calculus, Continuation-Passing Style, Logical Frameworks

Contents

List of Figures

1 Introduction

Continuation-Passing Style (CPS) λ-terms encode both evaluation order and sequencing order [13]. For example, consider the Direct Style (DS) λ-term

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

Evaluating it from left to right under call-by-value (CBV) amounts

1. to evaluate $f\,x$ — call the result v_1,
2. to evaluate $g\,x$ — call the result v_2, and
3. to apply v_1 to v_2 — call the result v_3.

CBV, left-to-right CPS transformation of this term yields

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

On the other hand, evaluating the DS λ-term above from right to left under CBV amounts

1. to evaluate $g\,x$ — call the result v_2,
2. to evaluate $f\,x$ — call the result v_1, and
3. to apply v_1 to v_2 — call the result v_3.

CBV, right-to-left CPS transformation of this λ term yields

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

In earlier work, the first author developed a textual inverse of the CPS transformation, *i.e.*, a "direct-style transformation" [2]. To this end, it was necessary to characterize CPS terms that correspond to the output of Plotkin's CPS transformation, after administrative reductions [3, 16]. However this characterization was not formally proven. The goal of this note is to prove it.

The proof has been implemented in Elf [10], a constraint logic programming language based on the logical framework LF [5]. In fact, it was the implementation that inspired the proof. LF turned out to be particularly suited for this problem, since two-level λ-terms and the CPS transformation can be encoded very naturally by using meta-level abstraction and application to model administrative reductions. This note thus also presents an excellent, albeit small, case study of machine-assisted proof discovery.

The rest of this note is organized as follows. Section 2 presents our starting point: the left-to-right CBV CPS transformation. We formulate it both as a function and as a judgment. Section 3 describes properties of CPS terms as produced by this CPS transformation: their BNF and the ordering of formal parameters of continuations. In Section 4, we prove that the output of the CPS transformation satisfies the ordering. Section 5 describes the implementation of the proof in Elf. Following a comparison with related work in Section 7, Section 8 concludes.

2 The CPS Transformation

The BNF of the pure λ-calculus reads as follows. We refer to this λ-calculus as *direct style* (DS) to distinguish it from the *continuation-passing style* (CPS) calculus introduced later.

$r \in$ 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 displays a one-pass CPS transformer for the pure call-by-value λ-calculus. This transformer is an optimized version of Plotkin's CPS transformer [13], derived in an earlier work [3]; it is slightly rephrased to match the syntactic domains.

These equations can be read as a two-level specification *à la* Nielson and Nielson [8]. Operationally,

- for any variable x and any expressions e, e_0, and e_1, $[x]e$ and $e_0(e_1)$ respectively correspond to functional abstractions and applications in the translation program (and define the so-called "administrative reductions"), and
- for any variable x and any expressions e, e_0, and e_1, $\lambda x.e$ and $e_0\, e_1$ respectively represent abstract-syntax constructors (to build the residual program).

Note that the types of the translations and continuations are meta-level types: The object calculus is untyped. We revisit these types in Section 5.2.

The CPS transformation can be reformulated with three judgments. A DS term r is transformed into a CPS term r' whenever the judgment

$$\vdash r \overset{\text{DRoot}}{\longrightarrow} r'$$

is satisfied. Given a continuation κ, a DS expression e is transformed into a CPS expression e' whenever the judgment

$$\vdash e \,;\, \kappa \overset{\text{DExp}}{\longrightarrow} e'$$

is satisfied. Finally, a DS trivial expression t is transformed into a CPS trivial expression t' whenever the judgment

$$\vdash t \overset{\text{DTriv}}{\longrightarrow} t'$$

is satisfied. The overall transformation is displayed in Figure 2.

NB: In the inference rule for applications, t_0 is "new", *i.e.*, the deduction of the left premise is parametric in t_0. This means that we can substitute an arbitrary trivial term t for t_0 in this derivation and obtain a derivation of $\vdash e_1 \,;\, [t_1]\, t\, t_1\, \lambda v.\kappa(v) \overset{\text{DExp}}{\longrightarrow} e_1'(t)$. This property is exploited crucially in the proof of Section 4.

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

where k and the v's are fresh variables.

Figure 1: The left-to-right, call-by-value CPS transformation formulated as a function

$$
\frac{\vdash e\,;\,[t]\,k\,t \overset{\text{DExp}}{\longrightarrow} e'}{\vdash e \overset{\text{DRoot}}{\longrightarrow} \lambda k.e'}
$$

$$
\frac{\vdash e_1\,;\,[t_1]\,t_0\,t_1\,\lambda v.\kappa(v) \overset{\text{DExp}}{\longrightarrow} e_1'(t_0) \qquad \vdash e_0\,;\,[t_0]\,e_1'(t_0) \overset{\text{DExp}}{\longrightarrow} e'}{\vdash e_0\,e_1\,;\,\kappa \overset{\text{DExp}}{\longrightarrow} e'}
\qquad
\frac{\vdash t \overset{\text{DTriv}}{\longrightarrow} t'}{\vdash t\,;\,\kappa \overset{\text{DExp}}{\longrightarrow} \kappa(t')}
$$

$$
\frac{}{\vdash x \overset{\text{DTriv}}{\longrightarrow} x}
\qquad
\frac{\vdash r \overset{\text{DRoot}}{\longrightarrow} r'}{\vdash \lambda x.r \overset{\text{DTriv}}{\longrightarrow} \lambda x.r'}
$$

Figure 2: The left-to-right, call-by-value CPS transformation formulated as a judgment

3 CPS Terms

We first specify the BNF of CPS terms as produced by the CPS transformation of Figures 1 and 2, and then we specify the occurrence conditions over the continuations and their formal parameters. Both specifications come from the earlier work on the DS transformation [2].

3.1 BNF of CPS terms

The BNF of CPS terms reads as follows. (NB: We distinguish between the original identifiers x coming from the DS term, and the fresh identifiers v and k introduced by $\mathcal{C}$.)

$r \in$ 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	
$v \in$ Var	— fresh parameters of continuations	
$k \in$ Cont	— fresh variables denoting continuations	

3.2 Occurrences of continuation parameters

The occurrence conditions over continuation parameters is simple: there is only one continuation at any point of a CPS term. This is captured in Figure 3 and proven in Appendix A.

CPS terms that do not satisfy the occurrence conditions over continuation parameters correspond to DS terms that use a control operator such as **call/cc**. This point is investigated elsewhere [4, 6].

3.3 Occurrences of formal parameters of continuations

The occurrence conditions over the formal parameters of continuations are reproduced in Figure 4. This figure should be read as follows. Given a CPS expression e occurring in the scope of formal parameters of continuations listed in the order of their declaration in a list ξ, the judgment

$$\xi \vdash^{\mathrm{CExp}}_{\mathrm{Var}} e$$

is satisfied whenever the variables listed in ξ and all the other formal parameters of continuations declared in e occur in a left-to-right fashion in e. (NB: $\bullet$ denotes the empty list.)

Similarly, given a trivial term t occurring in the scope of formal parameters of continuations listed in the order of declaration in ξ, the judgment

$$\xi \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} t \ ; \ \xi'$$

is satisfied whenever ξ' is a prefix of ξ and the remaining variables of ξ' occur in t in a left-to-right fashion.

Our goal here is to prove that transforming a DS term r with $\mathcal{C}$ (in Figures 1 and 2) yields a CPS term that satisfies the judgment

$$\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} \mathcal{C}[\![r]\!].$$

$$\frac{k \vdash^{\mathrm{CExp}}_{\mathrm{Cont}} e}{\vdash^{\mathrm{CRoot}}_{\mathrm{Cont}} \lambda k.e}$$

$$\frac{\vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} t_1 \qquad \vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} t_0 \qquad k \vdash^{\mathrm{CExp}}_{\mathrm{Cont}} e}{k \vdash^{\mathrm{CExp}}_{\mathrm{Cont}} t_0\, t_1\, \lambda v.e} \qquad \frac{\vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} t}{k \vdash^{\mathrm{CExp}}_{\mathrm{Cont}} k\, t}$$

$$\frac{}{\vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} x} \qquad \frac{\vdash^{\mathrm{CRoot}}_{\mathrm{Cont}} r}{\vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} \lambda x.r} \qquad \frac{}{\vdash^{\mathrm{CTriv}}_{\mathrm{Cont}} v}$$

Figure 3: Occurrences of continuation parameters in a CPS term

$$\frac{\bullet \vdash^{\mathrm{CExp}}_{\mathrm{Var}} e}{\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} \lambda k.e}$$

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

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

Figure 4: Ordering over formal parameters of continuations in a CPS term

NB: There is nothing wrong with CPS terms that do not satisfy the judgments of Figure 4. Simply, they specify another evaluation order or another sequencing order than the one captured in the CPS transformation of Figures 1 and 2. Therefore, they cannot be mapped back to direct style naïvely [2, 7].

4 The Proof

Globally, we are interested in proving that if $\vdash r \overset{\mathrm{DRoot}}{\longrightarrow} r'$ then $\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} r'$. Clearly, we cannot prove this inductively by itself since properties at the root of a term are defined in terms of the expressions it contains. The critical issue is the property of *continuations* we must prove (in the inductive conclusion) and require (in the inductive hypothesis) for the translation of expressions under a continuation. A continuation is a (meta-level) function from trivial terms to expressions, which suggests the method of logical relations [17]. The idea behind binary logical relations is to consider two functions related if they map related arguments to related results. In unary form: A function is valid if it maps valid arguments to valid results. This kind of definition is pervasive in the application of logical frameworks to meta-theoretic reasoning (*e.g.*, [9]). It works smoothly here.

Four notions of validity arise: for root terms, for trivial expressions, for serious expressions, and for continuations. In their definitions, we must account for the context ξ in which an expression might occur. For root terms, serious expressions, and trivial expressions, the notion of validity is derived directly from the property we are trying to prove; for continuations it arises from the considerations of logical relations as motivated above. We also streamline the definitions by considering separately the case of a trivial variable v, since such a variable is never the result of the translation of a trivial DS term (see Theorem 1 (3)).

Definition 1

(1) *r' is* valid *if* $\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} r'$.

(2) *e' is* ξ-valid *if* $\xi \vdash^{\mathrm{CExp}}_{\mathrm{Var}} e'$.

(3) *t' is* valid *if* $\xi \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} t' \,;\, \xi$ *for every* ξ.

(4) *κ is* ξ-valid *if*

  (a) $\xi, v \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa(v)$, *and*

  (b) $\xi \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa(t')$, *for any valid* t'.

This definition is more complex than it may appear at first, since it involves meta-level applications $\kappa(v)$ and $\kappa(t')$ and therefore, implicitly, substitution.

Theorem 1

(1) *If* $\vdash r \overset{\mathrm{DRoot}}{\longrightarrow} r'$ *then* r' *is valid.*

(2) *If* κ *is* ξ*-valid and* $\vdash e\ ;\ \kappa \stackrel{\mathrm{DExp}}{\longrightarrow} e'$ *then* e' *is* ξ*-valid.*

(3) *If* $\vdash t \stackrel{\mathrm{DTriv}}{\longrightarrow} t'$ *then* t' *is valid.*

Proof: By mutual induction on the derivations $\mathcal{R}$, $\mathcal{E}$, and $\mathcal{T}$ of $\vdash r \stackrel{\mathrm{DRoot}}{\longrightarrow} r'$, $\vdash e\ ;\ \kappa \stackrel{\mathrm{DExp}}{\longrightarrow} e'$, and $\vdash t \stackrel{\mathrm{DTriv}}{\longrightarrow} t'$, respectively.

Case $\mathcal{R} = \dfrac{\overset{\mathcal{E}}{\vdash e\ ;\ [t]\, k\, t \stackrel{\mathrm{DExp}}{\longrightarrow} e'}}{\vdash e \stackrel{\mathrm{DRoot}}{\longrightarrow} \lambda k.e'}$

Then $\kappa = [t]\, k\, t$ is $\bullet$-valid:

(a) $\dfrac{\overline{\bullet,\ v \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} v\ ;\ \bullet}}{\bullet \vdash^{\mathrm{CExp}}_{\mathrm{Var}} k\, v}$ holds, and

(b) $\dfrac{\bullet \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} t'\ ;\ \bullet}{\bullet \vdash^{\mathrm{CExp}}_{\mathrm{Var}} k\, t'}$ for any valid t'.

Hence, by induction hypothesis (2) on $\mathcal{E}$, $\bullet \vdash^{\mathrm{CExp}}_{\mathrm{Var}} e'$, and thus $\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} \lambda k.e'$.

Case $\mathcal{E} = \dfrac{\overset{\mathcal{E}_1(t_0)}{\vdash e_1\ ;\ [t_1]\, t_0\, t_1\, \lambda v.\kappa(v) \stackrel{\mathrm{DExp}}{\longrightarrow} e_1'(t_0)} \qquad \overset{\mathcal{E}_0}{\vdash e_0\ ;\ [t_0]\, e_1'(t_0) \stackrel{\mathrm{DExp}}{\longrightarrow} e'}}{\vdash e_0\, e_1\ ;\ \kappa \stackrel{\mathrm{DExp}}{\longrightarrow} e'}$

Assume κ is ξ-valid. We need to show that $\kappa_0 = [t_0]\, e_1'(t_0)$ is ξ-valid, since then $\xi \vdash^{\mathrm{CExp}}_{\mathrm{Var}} e'$ by induction hypothesis (2) on $\mathcal{E}_0$. Thus we need to show properties (a) and (b) for κ_0.

(a) We need $\xi,\ v_0 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa_0(v_0)$. Consider $\overset{\mathcal{E}_1(v_0)}{\vdash e_1\ ;\ [t_1]\, v_0\, t_1\, \lambda v.\kappa(v) \stackrel{\mathrm{DExp}}{\longrightarrow} e_1'(v_0)}$. We would like to show that

$$\kappa_1 = [t_1]\, v_0\, t_1\, \lambda v.\kappa(v)$$

is $\xi,\ v_0$-valid, since then $e_1'(v_0) = \kappa_0(v_0)$ is $\xi,\ v_0$-valid by induction hypothesis (2) on $\mathcal{E}_1(v_0)$. Therefore we need to consider the two cases of Definition 1(4).

(a) $\xi,\ v_0,\ v_1 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa_1(v_1)$. We derive this as follows:

$$\dfrac{\overline{\xi,\ v_0,\ v_1 \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} v_1\ ;\ \xi,\ v_0} \qquad \overline{\xi,\ v_0 \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} v_0\ ;\ \xi} \qquad \overset{\text{since } \kappa \text{ is } \xi\text{-valid}}{\xi,\ v \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa(v)}}{\xi,\ v_0,\ v_1 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} v_0\, v_1\, \lambda v.\kappa(v)}$$

(b) $\xi,\ v_0 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa(t_1')$, where t_1' is valid. This is established by the derivation

$$\dfrac{\overset{\text{since } t_1' \text{ is valid}}{\xi,\ v_0 \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} t_1'\ ;\ \xi,\ v_0} \qquad \overline{\xi,\ v_0 \vdash^{\mathrm{CTriv}}_{\mathrm{Var}} v_0\ ;\ \xi} \qquad \overset{\text{since } \kappa \text{ is } \xi\text{-valid}}{\xi,\ v \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa(v)}}{\xi,\ v_0,\ v_1 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} v_0\, t_1'\, \lambda v.\kappa(v)}$$

Thus κ_1 is $\xi,\ v_0$-valid. Therefore, by induction hypothesis on $\mathcal{E}_1(v_0)$,

$$\xi,\ v_0 \vdash^{\mathrm{CExp}}_{\mathrm{Var}} \kappa_0(v_0).$$

(b) We need $\xi \vdash^{\text{CExp}}_{\text{Var}} \kappa_0(t'_0)$ for any valid t'_0. Consider

$$\dfrac{\mathcal{E}_1(t'_0)}{\vdash e_1 \;;\; [t_1]\, t'_0\, t_1\, \lambda v.\kappa(v) \overset{\text{DExp}}{\longrightarrow} \underbrace{e'_1(t'_0)}_{= \kappa_0(t'_0)}}$$

We would like to show that

$$\kappa_1 = [t_1]\, t'_0\, t_1\, \lambda v.\kappa(v)$$

is ξ-valid, so we can apply the induction hypothesis to $\mathcal{E}_1(t'_0)$. Again, we need to consider the two clauses of Definition 1(4).

(a) $\xi,\, v_1 \vdash^{\text{CExp}}_{\text{Var}} \kappa_1(v_1)$. We derive this as follows:

$$\dfrac{\dfrac{}{\xi,\, v_1 \vdash^{\text{CTriv}}_{\text{Var}} v_1 \;;\; \xi} \qquad \begin{array}{c}\text{since } t'_0 \text{ is valid}\\ \xi \vdash^{\text{CTriv}}_{\text{Var}} t'_0 \;;\; \xi\end{array} \qquad \begin{array}{c}\text{since } \kappa \text{ is } \xi\text{-valid}\\ \xi,\, v \vdash^{\text{CExp}}_{\text{Var}} \kappa(v)\end{array}}{\xi,\, v_1 \vdash^{\text{CExp}}_{\text{Var}} t'_0\, v_1\, \lambda v.\kappa(v)}$$

(b) $\xi \vdash^{\text{CExp}}_{\text{Var}} \kappa_1(t'_1)$ for any valid t'_1. We construct:

$$\dfrac{\begin{array}{c}\text{since } t'_1 \text{ is valid}\\ \xi \vdash^{\text{CTriv}}_{\text{Var}} t'_1 \;;\; \xi\end{array} \qquad \begin{array}{c}\text{since } t'_0 \text{ is valid}\\ \xi \vdash^{\text{CTriv}}_{\text{Var}} t'_0 \;;\; \xi\end{array} \qquad \begin{array}{c}\text{since } \kappa \text{ is } \xi\text{-valid}\\ \xi,\, v \vdash^{\text{CExp}}_{\text{Var}} \kappa(v)\end{array}}{\xi \vdash^{\text{CExp}}_{\text{Var}} t'_0\, t'_1\, \lambda v.\kappa(v)}$$

Hence κ_1 is ξ-valid and thus $\xi \vdash^{\text{CExp}}_{\text{Var}} \underbrace{e'_1(t'_0)}_{= \kappa_0(t'_0)}$ by induction hypothesis (2) on $\mathcal{E}_1(t'_0)$.

Thus κ_0 is ξ-valid. Hence e' is valid by induction hypothesis (2) on $\mathcal{E}_0$.

Case $\mathcal{E} = \dfrac{\dfrac{\mathcal{T}}{\vdash t \overset{\text{DTriv}}{\longrightarrow} t'}}{\vdash t \;;\; \kappa \overset{\text{DExp}}{\longrightarrow} \kappa(t')}$.

By induction hypothesis (3) on $\mathcal{T}$, t' is valid. Since we assume that κ is ξ-valid, $\kappa(t')$ is also ξ-valid by clause (b) in Definition 1.

Case $\mathcal{T} = \dfrac{}{\vdash x \overset{\text{DTriv}}{\longrightarrow} x}$. Then $\dfrac{}{\xi \vdash^{\text{CTriv}}_{\text{Var}} x \;;\; \xi}$ is an axiom for any ξ.

Case $\mathcal{T} = \dfrac{\dfrac{\mathcal{R}}{\vdash r \overset{\text{DRoot}}{\longrightarrow} r'}}{\vdash \lambda x.r \overset{\text{DTriv}}{\longrightarrow} \lambda x.r'}$. Then we construct $\dfrac{\begin{array}{c}\text{by i.h. (1) on } \mathcal{R}\\ \vdash^{\text{CRoot}}_{\text{Var}} r'\end{array}}{\xi \vdash^{\text{CTriv}}_{\text{Var}} \lambda x.r' \;;\; \xi}$.

□

5 Implementation in Elf

In this section we show the implementations of the DS and CPS terms, CPS transformation, ordering, and the proof that the results of the CPS transformation are valid. Familiarity with the LF logical framework [5], its methodology, and it implementation in Elf [10] is assumed. Some implementation-specific details will be mentioned in the commentary.

5.1 Direct-style terms

Recall the information definition of direct-style (DS) terms in BNF form.

$$\begin{array}{rrcl} \text{DS (Root) Terms} & r & ::= & e \\ \text{DS (Serious) Expressions} & e & ::= & e_0\, e_1 \mid t \\ \text{DS Trivial Expressions} & t & ::= & x \mid \lambda x.r \end{array}$$

We only remark that the representation uses *higher-order abstract syntax* [11] to represent object-level abstractions, and that the natural inclusions (*e.g.*, every trivial expression is an expression) are modeled by explicit coercions (*e.g.*, `dtriv_dexp`).

```
droot : type. %name droot R
dexp  : type. %name dexp E
dtriv : type. %name dtriv T

dexp_droot : dexp -> droot.
dapp       : dexp -> dexp -> dexp.
dtriv_dexp : dtriv -> dexp.
dlam       : (dtriv -> droot) -> dtriv.
```

Note that `dlam` abstracts over an argument of type `dtriv`, thus encoding the fact that variables x are trivial expressions. The `%name` declarations indicate preferred variable names for syntactic classes, in case the Elf interpreter has to synthesize names (which is a frequent occurrence in during type reconstruction).

5.2 CPS terms

Recall the definition of continuation-passing style (CPS) terms in BNF form.

$$\begin{array}{rrcl} \text{CPS (Root) Terms} & r & ::= & \lambda k.e \\ \text{CPS (Serious) Expressions} & e & ::= & e_0\, e_1\, \lambda v.e \mid k\, t \\ \text{CPS Trivial Expressions} & t & ::= & x \mid \lambda x.r \mid v \end{array}$$

CPS terms are modelled using the same principles as DS terms, but they introduce a new consideration. The two-level CPS transformation from Section 2 shows that a continuation is best considered as a meta-level function which, when applied to a trival term, yields an expression. It therefore has type `ctriv -> cexp`. An abstraction over a continuation (as is necessary for a root term $\lambda k.e$) thus is a third-order construct! This is rare and indicates that we are exploiting the expressive power of the meta-language to a great extent.

```
croot : type. %name croot R
cexp  : type. %name cexp E
ctriv : type. %name ctriv T
% ccont : type = ctriv -> cexp. %name ccont K

rlam  : ((ctriv -> cexp) -> cexp) -> croot.
capp  : ctriv -> ctriv -> (ctriv -> cexp) -> cexp.
clam  : (ctriv -> croot) -> ctriv.
```

Note that Elf currently does not support definitions, so we must write the expanded version of the continuation type `ccont` by hand. It is inserted in the source only as a comment.

5.3 The CPS transformation

The judgments in Figure 2 can be easily transcribed into Elf. Just like the inference rules themselves, the corresponding declarations below should be understood schematically—the free variables are implicitly quantified. Elf's type reconstruction determines the most general type for the free variables in each declaration.

Instead of `d : A -> (B -> C)` we often use the form `d : C <- B <- A` to emphasize the operational interpretation of the declarations as a logic program (to solve `C` first solve `B` then `A`). In this case, the logic program transforms DS terms to CPS terms. The `%mode` pragmas establish the role of input (`+`) and output (`-`) arguments to a predicate. They are checked for consistency, thus providing operational correctness guarantees beyond type correctness. The `%lex` annotation postulates a termination ordering on the given arguments and modes which is checked by Elf. In this case we simply use the subterm ordering on the first argument of the three mutually recursive judgments.

```
cst_r : droot -> croot -> type.                   %name cst_r CR
cst_e : dexp -> (ctriv -> cexp) -> cexp -> type.  %name cst_e CE
cst_t : dtriv -> ctriv -> type.                   %name cst_t CT

%mode -cst_r +R -R'
%mode -cst_e +E +K -E'
%mode -cst_t +T -T'
%lex {R E T}

cst_r_dexp : cst_r (dexp_droot E) (rlam E')
              <- ({k:ctriv -> cexp} cst_e E k (E' k)).

cst_e_dapp :
   cst_e (dapp E0 E1) K E'
      <- ({t0:ctriv} cst_e E1 ([t1:ctriv] capp t0 t1 K) (E1' t0))
      <- cst_e E0 ([t0:ctriv] E1' t0) E'.

cst_e_dtriv : cst_e (dtriv_dexp T) K (K T')
               <- cst_t T T'.

cst_t_dlam : cst_t (dlam R) (clam R')
              <- ({x:dtriv} {x':ctriv} cst_t x x' -> cst_r (R x) (R' x')).
```

The left premise of the rule for applications $e_0\, e_1$ is required to be parametric in t_0. This is represented by a dependently typed function from t_0 to a derivation of

$$\vdash e_1 \,;\, [t_1]\, t_0\, t_1\, \lambda v.\kappa(v) \overset{\text{DExp}}{\longrightarrow} e_1'(t_0).$$

In Elf's concrete syntax this type is written as

```
{t0:ctriv} cst_e E1 ([t1:ctriv] capp t0 t1 K) (E1' t0)
```

Note that we have silently η-reduced $\lambda v.\kappa(v)$ and simply written `K`. This is a matter of style and efficiency, but not essential, since the definitional equality of the Elf meta-language is $\beta\eta$-conversion.

5.4 Ordering over parameters of continuations

In order to describe the ordering over parameters of continuations, we require a notion of stack which is easily defined. The `%infix` declaration makes '`,`' a left-associative infix operator with an (arbitrary) binding strength of 10.

```
stack : type. %name stack Xi
dot : stack.
,   : stack -> ctriv -> stack.  %infix left 10 ,
```

The three mutually recursive judgments regarding variable ordering are easily translated into Elf. Note that the cases concerning variables x, v and k must be given wherever such variables are introduced, rather than globally. This is a consequence of the representation technique of higher-order abstract syntax.

```
ord_r : croot -> type.                    %name ord_r OR
ord_e : stack -> cexp -> type.            %name ord_e OE
ord_t : stack -> ctriv -> stack -> type.  %name ord_t OT

%mode -ord_r +R
%mode -ord_e +Xi +E
%mode -ord_t +Xi' +T -Xi''
%lex {R E T}

ord_r_rlam : ord_r (rlam E)
              <- ({k:ctriv -> cexp}
                    ({Xi:stack} {T:ctriv}
                       ord_e Xi (k T) <- ord_t Xi T dot)
                    -> ord_e dot (E k)).

ord_e_capp : ord_e Xi (capp T0 T1 E)
              <- ord_t Xi T1 Xi1
              <- ord_t Xi1 T0 Xi0
              <- ({v:ctriv}
                    ({Xi':stack} ord_t (Xi' , v) v Xi')
                    -> ord_e (Xi0 , v) (E v)).

ord_t_clam : ({Xi:stack} ord_t Xi (clam R) Xi)
              <- ({x:ctriv}
                    ({Xi':stack} ord_t Xi' x Xi')
                    -> ord_r (R x)).
```

5.5 The proof

The informal proof in Section 4 that continuation parameters obey a stack-like discipline can be translated into Elf using the technique of *higher-level judgments* (see, for example, [12]). Our (constructive) proof may be seen as containing an algorithm for computing a derivation $\mathcal{R}'$ of $\vdash^{\mathrm{CRoot}}_{\mathrm{Var}} r'$ from a derivation $\mathcal{R}$ of $\vdash r \xrightarrow{\mathrm{DRoot}} r'$. In Elf, this algorithm is implemented as a logic program for transforming $\mathcal{R}$ into $\mathcal{R}'$; declaratively it is a higher-level judgment relating derivations $\mathcal{R}$ and $\mathcal{R}'$. Properties of these higher-level judgments such as termination can then be established automatically.

In order to match the definition of the CPS transformation closely, our formalization does not use explicit definitions of validity except for continuations κ, which would otherwise be unwieldy.

```
valid_k : stack -> (ctriv -> cexp) -> type.

%mode -valid_k +Xi +K
%lex K

vld_k : valid_k Xi K
         <- ({v:ctriv}
               ({Xi':stack} ord_t (Xi' , v) v Xi')
               -> ord_e (Xi , v) (K v))
         <- ({t':ctriv}
               ({Xi:stack} ord_t Xi t' Xi)
               -> ord_e Xi (K t')).
```

The proof is implemented by three mutually recursive higher-level judgments for root terms, expressions, and trivial expressions. Each clause corresponds to one case of the informal proof. Each appeal to an induction hypothesis appears as a recursive call.

```
proof_r : cst_r R R' -> ord_r R' -> type.
proof_e : cst_e E K E' -> valid_k Xi K -> ord_e Xi E' -> type.
proof_t : cst_t T T' -> ({Xi:stack} ord_t Xi T' Xi) -> type.

%mode -proof_r +CR -OR
%mode -proof_e +CE +VK -OE
%mode -proof_t +CT -OT
%lex {CR CE CT}

pf_r : proof_r (cst_r_dexp CE) (ord_r_rlam OE)
        <- ({k:ctriv -> cexp}
              {ok : {Xi:stack} {T:ctriv} ord_e Xi (k T) <- ord_t Xi T dot}
              proof_e (CE k)
              (vld_k
                 ([t':ctriv] [CT:{Xi:stack} ord_t Xi t' Xi]
                    ok dot t' (CT dot))
                 ([v:ctriv] [CT:{Xi':stack} ord_t (Xi' , v) v Xi']
                    ok (dot , v) v (CT dot)))
              (OE k ok)).

pf_e_dapp : proof_e (cst_e_dapp CE0 CE1) (vld_k _ OE) OE'
             <- ({v0 : ctriv} {OT0 : {Xi':stack} ord_t (Xi' , v0) v0 Xi'}
                   proof_e (CE1 v0)
                   (vld_k
                      ([t1:ctriv] [OT1:{Xi':stack} ord_t Xi' t1 Xi']
                         ord_e_capp OE (OT0 Xi) (OT1 (Xi , v0)))
                      ([v1:ctriv] [OT1:{Xi':stack} ord_t (Xi' , v1) v1 Xi']
                         ord_e_capp OE (OT0 Xi) (OT1 (Xi , v0))))
                   (VE1'V v0 OT0))
             <- ({t0 : ctriv} {OT0 : {Xi':stack} ord_t Xi' t0 Xi'}
                   proof_e (CE1 t0)
                   (vld_k
                      ([t1:ctriv] [OT1:{Xi':stack} ord_t Xi' t1 Xi']
```

```
                        ord_e_capp OE (OT0 Xi) (OT1 Xi))
                     ([v1:ctriv] [OT1:{Xi':stack} ord_t (Xi' , v1) v1 Xi']
                        ord_e_capp OE (OT0 Xi) (OT1 Xi)))
                  (VE1'T t0 OT0))
             <- proof_e CE0 (vld_k VE1'T VE1'V) OE'.

pf_e_dtriv : proof_e (cst_e_dtriv CT) (vld_k OE _) (OE T' OT)
              <- proof_t CT OT.

pf_t_dlam : proof_t (cst_t_dlam CR) (ord_t_clam OR)
             <- ({x:dtriv} {x':ctriv}
                   {CT: cst_t x x'}
                   {OT:{Xi':stack} ord_t Xi' x' Xi'}
                   proof_t CT OT
                   -> proof_r (CR x x' CT) (OR x' OT)).
```

From the implementation above it is actually quite easy (with a little experience) to reconstruct the informal proof.

The proof of the property of occurrences of continuations k themselves (see Figure 3) can also easily be represented in the same style. It can be found in Appendix A.

5.6 An example

We now reconsider the direct-style term from Section 1.

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

Under appropriate declarations for f and g as variables, this term is represented in Elf by

```
(dexp_droot
   (dtriv_dexp
      (dlam [x:dtriv]
         dexp_droot (dapp (dapp (dtriv_dexp f) (dtriv_dexp x))
                          (dapp (dtriv_dexp g) (dtriv_dexp x))))))
: droot.
```

It is rather lengthy due to the coercions, but we could easily write a judgment to insert appropriate coercions into pure λ-term. In order to translate this we may pose the following query.

```
CR:
cst_r (dexp_droot
         (dtriv_dexp
            (dlam [x:dtriv]
               dexp_droot (dapp (dapp (dtriv_dexp f) (dtriv_dexp x))
                                (dapp (dtriv_dexp g) (dtriv_dexp x))))))
R.
```

which yields the CPS term `R` (eliding the derivation `CR`)

```
R =
   rlam [k:ctriv -> cexp]
      k (clam [x':ctriv] rlam [k1:ctriv -> cexp]
            capp f' x' ([t01:ctriv] capp g' x' ([t1:ctriv] capp t01 t1 k1))),
CR = ...
```

Modulo variable names, this corresponds to

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

The omitted term CR represents the derivation of the judgment

$$\vdash \lambda x.f\,x\,(g\,x) \stackrel{\text{DRoot}}{\longrightarrow} \lambda k.k\,(\lambda x.\lambda k.f\,x\,\lambda v_1.g\,x\,\lambda v_2.v_1\,v_2\,\lambda v_3.k\,v_3)$$

which was constructed by the Elf interpreter in answer to the first query. We can apply the implementation of the meta-theory to translate CR into a derivation showing that the conditions on occurrences of continuation parameters are satisfied in this example, that is, into a derivation of

$$\vdash^{\text{CRoot}}_{\text{Var}} \lambda k.k\,(\lambda x.\lambda k.f\,x\,\lambda v_1.g\,x\,\lambda v_2.v_1\,v_2\,\lambda v_3.k\,v_3).$$

The query is the following. The first argument to proof_r is the derivation CR elided above.

```
proof_r
(cst_r_dexp [k:ctriv -> cexp]
    cst_e_dtriv
       (cst_t_dlam [x:dtriv] [x'1:ctriv] [CT:cst_t x x'1]
           cst_r_dexp [k1:ctriv -> cexp]
           cst_e_dapp
              (cst_e_dapp (cst_e_dtriv cst_f) ([t0:ctriv] cst_e_dtriv CT))
              ([t0:ctriv]
                  cst_e_dapp (cst_e_dtriv cst_g) ([t01:ctriv] cst_e_dtriv CT))))
OR.
```

We know that a query of this form will always succeed. In this case it produces the substitution

```
OR =
   ord_r_rlam [k:ctriv -> cexp]
      [ok:{Xi:stack} {T:ctriv} ord_t Xi T dot -> ord_e Xi (k T)]
      ok dot
         (clam [x':ctriv] rlam [k1:ctriv -> cexp]
             capp f' x' ([t0:ctriv] capp g' x' ([t1:ctriv] capp t0 t1 k1)))
         (ord_t_clam
             ([x'1:ctriv] [OT:{Xi':stack} ord_t Xi' x'1 Xi']
                 ord_r_rlam [k1:ctriv -> cexp]
                 [ok1:{Xi:stack} {T:ctriv} ord_t Xi T dot -> ord_e Xi (k1 T)]
                 ord_e_capp
                    ([v0:ctriv] [OT01:{Xi':stack} ord_t (Xi' , v0) v0 Xi']
                        ord_e_capp
                           ([v1:ctriv] [OT1:{Xi':stack} ord_t (Xi' , v1) v1 Xi']
                               ord_e_capp
                                  ([v:ctriv]
                                      [CT:{Xi':stack} ord_t (Xi' , v) v Xi']
                                      ok1 (dot , v) v (CT dot))
                                  (OT01 dot) (OT1 (dot , v0)))
                           (ord_t_g (dot , v0)) (OT (dot , v0)))
                    (ord_t_f dot) (OT dot))
             dot).
```

which shows that the CPS term above satisfies the ordering criterion.

$$
\begin{array}{rcl}
\mathcal{D}^{\mathrm{CRoot}} & : & \mathrm{CRoot} \rightarrow \mathrm{DRoot} \\
\mathcal{D}^{\mathrm{CRoot}}[\![\lambda k.e]\!] & = & \mathcal{D}^{\mathrm{CExp}}[\![e]\!] \\
\\
\mathcal{D}^{\mathrm{CExp}} & : & \mathrm{CExp} \rightarrow \mathrm{DExp} \\
\mathcal{D}^{\mathrm{CExp}}[\![t_0\, t_1\, \lambda v.e]\!] & = & \mathcal{D}^{\mathrm{CExp}}[\![e]\!]\ [v := \mathcal{D}^{\mathrm{CTriv}}[\![t_0]\!]\ \mathcal{D}^{\mathrm{CTriv}}[\![t_1]\!]\,] \\
\mathcal{D}^{\mathrm{CExp}}[\![k\, t]\!] & = & \mathcal{D}^{\mathrm{CTriv}}[\![t]\!] \\
\\
\mathcal{D}^{\mathrm{CTriv}} & : & \mathrm{CTriv} \rightarrow \mathrm{DExp} \\
\mathcal{D}^{\mathrm{CTriv}}[\![x]\!] & = & x \\
\mathcal{D}^{\mathrm{CTriv}}[\![\lambda x.r]\!] & = & \lambda x.\mathcal{D}^{\mathrm{CRoot}}[\![r]\!] \\
\mathcal{D}^{\mathrm{CTriv}}[\![v]\!] & = & v
\end{array}
$$

Figure 5: The call-by-value DS transformation formulated as a function and using substitutions

6 The Direct-Style Transformation

Having formalized and proven the occurrences of continuation parameters in CPS terms, we can now show the transformation from a CPS term back to direct style. Note that this transformation only applies to terms satisfying occurrence and ordering conditions.

The following implementation uses substitution (see Figure 5). An implementation that uses a stack ξ without explicitly relying on substitution is also possible (see Figures 6 and 7).

```
dst_r : croot -> droot -> type.
dst_e : cexp -> dexp -> type.
dst_t : ctriv -> dexp -> type.

%mode -dst_r +R -R'
%mode -dst_e +E -E'
%mode -dst_t +T -T'
%lex {R E T}

dst_r_rlam : dst_r (rlam E) (dexp_droot E')
              <- ({k:ctriv -> cexp}
                    ({T:ctriv} {E:dexp} dst_e (k T) E <- dst_t T E)
                    -> dst_e (E k) E').

dst_e_capp : dst_e (capp T0 T1 ([v:ctriv] E v)) E'
              <- dst_t T0 E0
              <- dst_t T1 E1
              <- ({v:ctriv} dst_t v (dapp E0 E1) -> dst_e (E v) E').

dst_t_clam : dst_t (clam R) (dtriv_dexp (dlam R'))
              <- ({x:ctriv} {x':dtriv}
                    dst_t x (dtriv_dexp x')
                    -> dst_r (R x) (R' x')).
```

$$
\begin{array}{rcl}
\mathcal{D}^{\mathrm{CRoot}} & : & \mathrm{CRoot} \rightarrow \mathrm{DRoot} \\
\mathcal{D}^{\mathrm{CRoot}}[\![\lambda k.e]\!] & = & \mathcal{D}^{\mathrm{CExp}}[\![e]\!]\,\bullet \\
\\
\mathcal{D}^{\mathrm{CExp}} & : & \mathrm{CExp} \rightarrow \mathrm{List}(\mathrm{DExp}) \rightarrow \mathrm{DExp} \\
\mathcal{D}^{\mathrm{CExp}}[\![t_0\,t_1\,\lambda v.e]\!]\,\xi & = & \mathrm{let}\ \langle e_1';\,\xi_1\rangle = \mathcal{D}^{\mathrm{CTriv}}[\![t_1]\!]\,\xi \\
 & & \mathrm{in}\ \mathrm{let}\ \langle e_0';\,\xi_0\rangle = \mathcal{D}^{\mathrm{CTriv}}[\![t_0]\!]\,\xi_1 \\
 & & \quad \mathrm{in}\ \mathcal{D}^{\mathrm{CExp}}[\![e]\!]\,(\xi_0,\,e_0'\,e_1') \\
\mathcal{D}^{\mathrm{CExp}}[\![k\,t]\!]\,\xi & = & \mathrm{let}\ \langle e';\,\bullet\rangle = \mathcal{D}^{\mathrm{CTriv}}[\![t]\!]\,\xi \\
 & & \mathrm{in}\ e' \\
\\
\mathcal{D}^{\mathrm{CTriv}} & : & \mathrm{CTriv} \rightarrow \mathrm{List}(\mathrm{DExp}) \rightarrow (\mathrm{DExp} \times \mathrm{List}(\mathrm{DExp})) \\
\mathcal{D}^{\mathrm{CTriv}}[\![x]\!]\,\xi & = & \langle x;\,\xi\rangle \\
\mathcal{D}^{\mathrm{CTriv}}[\![\lambda x.r]\!]\,\xi & = & \langle \lambda x.\mathcal{D}^{\mathrm{CRoot}}[\![r]\!];\,\xi\rangle \\
\mathcal{D}^{\mathrm{CTriv}}[\![v]\!]\,\langle \xi;\,e'\rangle & = & \langle e';\,\xi\rangle
\end{array}
$$

where "let $x = e$ in b" abbreviates "$([x]b)(e)$" and thus denotes an administrative reduction.

Figure 6: The call-by-value DS transformation formulated as a function and using a stack

$$
\frac{\bullet \vdash e \overset{\mathrm{CExp}}{\longrightarrow} e'}{\vdash \lambda k.e \overset{\mathrm{CRoot}}{\longrightarrow} e'}
$$

$$
\frac{\xi \vdash t_1 \overset{\mathrm{CTriv}}{\longrightarrow} e_1';\,\xi_1 \qquad \xi_1 \vdash t_0 \overset{\mathrm{CTriv}}{\longrightarrow} e_0';\,\xi_0 \qquad \xi_0,\,e_0'\,e_1' \vdash e \overset{\mathrm{CExp}}{\longrightarrow} e'}{\xi \vdash t_0\,t_1\,\lambda v.e \overset{\mathrm{CExp}}{\longrightarrow} e'}
\qquad
\frac{\xi \vdash t \overset{\mathrm{CTriv}}{\longrightarrow} e';\,\bullet}{\xi \vdash k\,t \overset{\mathrm{CExp}}{\longrightarrow} e'}
$$

$$
\frac{}{\xi \vdash x \overset{\mathrm{CTriv}}{\longrightarrow} x;\,\xi}
\qquad
\frac{\vdash r \overset{\mathrm{CRoot}}{\longrightarrow} r'}{\xi \vdash \lambda x.r \overset{\mathrm{CTriv}}{\longrightarrow} \lambda x.r';\,\xi}
\qquad
\frac{}{\xi,\,e' \vdash v \overset{\mathrm{CTriv}}{\longrightarrow} e';\,\xi}
$$

Figure 7: The call-by-value DS transformation formulated as a judgment and using a stack

7 Related Work

The structure of CPS terms has been little investigated. Most authors (*e.g.*, Wand and Oliva [18]) implicitly rely on conformant CPS terms to run them on a stack machine.

In their work on reasoning about CPS programs, Sabry and Felleisen also rely on the unicity of continuations parameters in the pure λ-calculus [14, 15].

In their work on separating stages in the CPS transformation [7], Lawall and Danvy noticed that the sequencing order encoded in CPS terms is accounted for by the occurrences of parameters of continuations. In his work on the DS transformation [2], Danvy characterized the ordering of Figure 4, but did not prove it formally. During spring 1993, Danvy and Pfenning carried out the work reported here. Later, in her PhD work on the inverseness of the CPS and the DS transformations, Lawall independently proved by hand a similar ordering [6, Appendix A.1.1].

8 Conclusion and Issues

We have formalized and proven the occurrences of continuation parameters and of formal parameters of continuations in CPS terms. This new knowledge about continuations parameters in CPS terms can enable their more efficient implementation. For example, the transformation of conforming CPS terms back to direct style can be implemented using a stack to carry out substitutions (see Figures 6 and 7). This new formulation also makes it simpler to prove that the CPS and the DS transformations are inverses of each other [6] and to automate this proof.

The implementation in Elf is small but non-trivial. It captures the computational content of the translations and the meta-theoretic reasoning in a declarative, yet executable way. The framework is built around the notions of substitution and and meta-level function, which leads to a very elegant and direct encoding. This representation is unusual in that it requires third-order constants (since it abstracts over continuations), thus exemplifying a new technique for representing deductive systems in LF interesting in its own right. Since the encoding suggested the proof technique, this paper demonstrates, on a small scale, the value of a logical framework as a conceptual tool in the study of the theory of programming languages.

A Occurrences of Continuations Parameters

Here we present the implementation of the occurrence condition on continuations parameters in CPS terms resulting from a CPS transformation (see Figure 3). Again, we use a third-order judgment.

```
occ_r: croot -> type.                              %name occ_r KR
occ_e: ((ctriv -> cexp) -> cexp) -> type.          %name occ_e KE
occ_t: ctriv -> type.                              %name occ_t KT

%mode -occ_r +R
%mode -occ_e +E
%mode -occ_t +T
%lex {R E T}
```

```
occ_r_rlam: occ_r (rlam E)
             <- occ_e E.

occ_e_capp: occ_e ([k:ctriv -> cexp] capp T0 T1 ([v:ctriv] (E k v)))
             <- occ_t T0
             <- occ_t T1
             <- ({v:ctriv}
                   occ_t v
                   -> occ_e ([k:ctriv -> cexp] (E k v))).

occ_e_cret: occ_e ([k:ctriv -> cexp] k T)
             <- occ_t T.

occ_t_clam: occ_t (clam R)
             <- ({x:ctriv}
                   occ_t x
                   -> occ_r (R x)).

%mode -occ_k +K
%lex K

occ_k: ((ctriv -> cexp) -> (ctriv -> cexp)) -> type.  %name occ_k KK

occ_k_k   : occ_k K
             <- ({t:ctriv}
                   occ_t t
                   -> occ_e ([k:ctriv -> cexp] K k t)).
```

Next is the implementation of the proof that the CPS transformation of DS terms yields CPS terms that satisfy the occurrence conditions of continuations parameters.

```
kproof_r : cst_r R R' -> occ_r R' -> type.
kproof_e : ({k:ctriv -> cexp} cst_e E (K k) (E' k))
            -> occ_k K -> occ_e E' -> type.
kproof_t : cst_t T T' -> occ_t T' -> type.

%mode -kproof_r +CR -KR
%mode -kproof_e +CE +KK -KE
%mode -kproof_t +CT -KT
%lex {CR CE CT}

kproof_r_dexp : kproof_r (cst_r_dexp CE) (occ_r_rlam KE)
                 <- kproof_e CE (occ_k_k [t:ctriv] [KT:occ_t t] occ_e_cret KT)
                    KE.

kproof_e_dapp : kproof_e ([k:ctriv -> cexp] cst_e_dapp (CE0 k) (CE1 k))
                 (occ_k_k KE') KE
                 <- ({t0:ctriv} {KT0:occ_t t0}
                       kproof_e ([k] CE1 k t0)
                       (occ_k_k [t1:ctriv] [KT1:occ_t t1]
                          occ_e_capp KE' KT1 KT0)
                       (KE1 t0 KT0))
                 <- kproof_e CE0 (occ_k_k KE1) KE.
```

```
kproof_e_dtriv : kproof_e ([k:ctriv -> cexp] cst_e_dtriv CT) (occ_k_k KE')
                   (KE' T' KT)
                   <- kproof_t CT KT.

kproof_t_dlam : kproof_t (cst_t_dlam CR) (occ_t_clam KR)
                  <- ({x:dtriv} {x':ctriv}
                        {Cx: cst_t x x'} {Kx':occ_t x'}
                        kproof_t Cx Kx' -> kproof_r (CR x x' Cx) (KR x' Kx')).
```

References

[1] 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.

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

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

[4] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [1], pages 299–310.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in the proceedings of the First IEEE Symposium on Logic in Computer Science, pages 194–204, June 1987.

[6] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, July 1994.

[7] 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.

[8] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[9] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[10] 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.

[11] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[12] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

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

[14] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [1], pages 288–298.

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

[16] 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.

[17] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[18] Mitchell Wand and Dino Oliva. Proving the correctness of storage representations. In Clinger [1], pages 151–160.