Three Steps for the CPS Transformation *
(detailed abstract)

Olivier Danvy
Department of Computing and Information Sciences
Kansas State University †
danvy@cis.ksu.edu

December 1991

Abstract

Transforming a λ-term into continuation-passing style (CPS) might seem mystical at first, but in fact it can be characterized by three separate aspects:

- The values of all *intermediate* applications are given a name.
- The evaluation of these applications is sequentialized based on a *traversal* of their syntax tree. This traversal mimics the reduction strategy.
- The resulting term is equipped with a *continuation* — a λ-abstraction whose application to intermediate values yields the final result of the whole evaluation.

The first point is fulfilled using the uniform naming mechanism of λ-abstraction (Church encoding), which explains why continuations are represented as functions. The second point justifies why CPS terms are evaluation-order independent — their evaluation order is determined by the syntax tree traversal of the CPS transformation. The third point captures the essence of the CPS transformation.

We have staged Fischer and Plotkin's original CPS transformer according to the three points above. For call-by-value λ-terms, the tree traversal is *post-order*. The method is applicable to other evaluation orders, *e.g.*, call-by-value with evaluation from right to left, or call-by-name.

This staging originates in the need for a clearer formulation of the CPS transformation, suitable for deriving its inverse.

Keywords

λ-calculus, abstract syntax trees, continuation-passing style transformation, Church encoding.

*Technical report CIS-92-02, Department of Computing and Information Sciences, Kansas State University. Revised version (February 14, 1992).

†Manhattan, KS 66506, USA. Phone: (913) 532-6350. FAX: (913) 532-7353. This work was partly supported by NSF under grant CCR-9102625.

1 Introduction

1.1 Background

Here is the one-pass version of Fischer & Plotkin's CPS transformation [8, 12] as we derived it in an earlier work [5, 6]. This translation yields terms without extraneous redexes, in one pass, thereby composing Fischer's transformation with Plotkin's colon-translation [8, 12, 6].

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
\llbracket \lambda\, x\, .\, M \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\underline{\lambda}x.\underline{\lambda}k.\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.\underline{@}k\, m)) \\
\llbracket @\, M\; N \rrbracket & = & \overline{\lambda}\kappa.\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.\overline{@}\llbracket N \rrbracket\,(\overline{\lambda}n.\underline{@}(\underline{@}m\, n)\,(\underline{\lambda}a.\overline{@}\kappa\, a)))
\end{array}
$$

where k is a fresh variable.

These equations can be read as a two-level specification *à la* Nielson and Nielson [11]. Operationally, the overlined λ's and @'s correspond to functional abstractions and applications in the translation program, while only the underlined occurrences represent abstract-syntax constructors.

The result of transforming a term M into CPS in an arbitrary context (represented by a continuation) is given by

$$\underline{\lambda}k.\overline{@}\llbracket M \rrbracket\,(\overline{\lambda}m.\underline{@}k\, m)$$

where k is a fresh variable.

1.2 Motivation

Even though the CPS transformation can be expressed in one pass, it still looks counter-intuitive. In an earlier work [5, 6], we expressed it in direct style using control abstractions, but though more concise the equations still remain enigmatic.

In a more recent work on the Direct Style (DS) transformation [4], we needed to express the CPS transformation in a form amenable to proving that the CPS and the DS transformations are inverses of each other. We were lead to staging the CPS transformation in a most enlightening way. The goal of this paper is to report this insight.

1.3 Illustration

Let us transform the following direct style term into CPS in three steps.

$$\lambda\, x\, .\, @\,(@\, f\; x)\;(@\, g\;(@\, h\; x))$$

To this effect, let us first name all intermediate values with fresh variables, using the **let** special form as a naming device (*cf.* Section 2). Intermediate values are produced by sub-terms that are applications. There are four of them here.

$$
\begin{array}{l}
\lambda\, x\, .\, \mathbf{let}\ v_0 = @\,(\mathbf{let}\ v_1 = @\, f\; x\ \mathbf{in}\ v_1) \\
\qquad\qquad\qquad\quad (\mathbf{let}\ v_2 = @\, g\;(\mathbf{let}\ v_3 = @\, h\; x\ \mathbf{in}\ v_3)\ \mathbf{in}\ v_2) \\
\qquad \mathbf{in}\ v_0
\end{array}
$$

As a second step (*cf.* Section 3), let us sequentialize the **let** expressions. This is possible because all the variables that have been introduced are fresh and therefore there are no risks of name clashes.

$$
\begin{array}{l}
\lambda x\,.\;\mathbf{let}\;v_1 = @\,f\,x \\
\quad\;\mathbf{in}\;\mathbf{let}\;v_3 = @\,h\,x \\
\quad\quad\;\mathbf{in}\;\mathbf{let}\;v_2 = @\,g\,v_3 \\
\quad\quad\quad\;\mathbf{in}\;\mathbf{let}\;v_0 = @\,v_1\,v_2 \\
\quad\quad\quad\quad\;\mathbf{in}\;v_0
\end{array}
$$

Finally (*cf.* Section 4), let us introduce a continuation for each function call, making its formal parameter coincide with the formal parameter of the corresponding **let** expression.

$$
\begin{array}{l}
\lambda k\,.\;@\,k \\
\quad\quad \lambda x\,.\,\lambda k\,.\;@\,(@\,f\,x) \\
\quad\quad\quad\quad \lambda v_1\,.\;@\,(@\,h\,x) \\
\quad\quad\quad\quad\quad\quad \lambda v_3\,.\;@\,(@\,g\,v_3) \\
\quad\quad\quad\quad\quad\quad\quad\quad \lambda v_2\,.\;@\,(@\,v_1\,v_2) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda v_0\,.\,@\,k\,v_0
\end{array}
$$

This CPS term (modulo renaming) is the one yielded by Fischer & Plotkin's CPS transformer for λ_v-terms. In fact, the composition of these three steps is equivalent to Fischer & Plotkin's CPS transformation.

1.4 Overview

The rest of this paper is organized as follows. Sections 2, 3, and 4 specify the three steps of the CPS transformation: naming intermediate values; sequentializing intermediate evaluations; and introducing continuations. Section 5 points out how Fischer & Plotkin's CPS transformer captures our two first steps into one, in a way that is compositional with respect to the source term and uses a functional accumulator. In Section 6, we specialize Fischer & Plotkin's CPS transformer to a restricted source syntax where the evaluation of all sub-expressions has already been sequentialized, and we point out how this specialized transformation coincides with our third step.

2 Naming Intermediate Values

Here is the BNF of direct-style λ-terms:

$$e \;::=\; x \;\mid\; \lambda x\,.\,e \;\mid\; @\,e_0\,e_1$$

Under call-by-value (applicative order), variables and abstractions are evaluated in one step, whereas applications need more evaluation steps: applications create intermediate values. For this reason, let us distinguish between variables and abstractions, and applications. This yields the following equivalent BNF of direct-style expressions.

$$
\begin{array}{rcl}
e & ::= & t \mid s \\
t & ::= & x \mid \lambda x\,.\,e \\
s & ::= & @\,e_0\,e_1
\end{array}
$$

where t and s respectively stand for "trivial" and "serious" — following Reynolds's terminology [13].

Evaluating a serious expression creates an intermediate value. The following transformation makes these intermediate values explicit by wrapping each application in a **let** expression.

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda\, x\,.\, e]\!] & = & \lambda\, x\,.\,[\![e]\!] \\
[\![@\, e_0\; e_1]\!] & = & \mathbf{let}\; v = @\,[\![e_0]\!]\;[\![e_1]\!]\;\mathbf{in}\; v
\end{array}
$$

where v is a fresh variable.

The semantics of a **let** expression is the usual strict one (we are working under call-by-value for now). The transformation is trivially correct.

This first transformation can be characterized as follows. For all λ-abstractions, all applications in the source term are now isolated in **let** expressions.

For example, the term

$$\lambda\, g\,.\,\lambda\, x\,.\,@\,(@\, g\; x)\;(\lambda\, z\,.\, z)$$

gets transformed into

$$\lambda\, g\,.\,\lambda\, x\,.\,\mathbf{let}\; v_0 = @\,(\mathbf{let}\; v_1 = @\, g\; x\;\mathbf{in}\; v_1)\;(\lambda\, z\,.\, z)\;\mathbf{in}\; v_0$$

3 Sequentializing Intermediate Evaluations

At this stage, here is the BNF for λ-terms as they are produced by the first step (*cf.* Section 2).

$$
\begin{array}{lcl}
e & ::= & t \mid s \\
t & ::= & x \mid \lambda\, x\,.\, e \\
s & ::= & \mathbf{let}\; v = @\, e_0\; e_1\;\mathbf{in}\; v
\end{array}
$$

Following the intuition of Section 1.3, let us sequentialize the embedded **let** expressions. To this effect, we rewrite these λ-terms until they satisfy the following BNF.

$$
\begin{array}{lcl}
e & ::= & t \mid s \\
t & ::= & x \mid \lambda\, x\,.\, e \\
s & ::= & \mathbf{let}\; v = @\, t_0\; t_1\;\mathbf{in}\; e
\end{array}
$$

The following rewrite system "flattens" embedded **let** expressions.

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\lambda\, x\,.\, e]\!] & = & \lambda\, x\,.\,[\![e]\!] \\
[\![\mathbf{let}\; v = @\,(\mathbf{let}\; w = e_0\;\mathbf{in}\; e_1)\; e_2\;\mathbf{in}\; e_3]\!] & = & [\![\mathbf{let}\; w = e_0\;\mathbf{in}\;\mathbf{let}\; v = @\, e_1\; e_2\;\mathbf{in}\; e_3]\!] \\
[\![\mathbf{let}\; v = @\, t_0\;(\mathbf{let}\; w = e_1\;\mathbf{in}\; e_2)\;\mathbf{in}\; e_3]\!] & = & [\![\mathbf{let}\; w = e_1\;\mathbf{in}\;\mathbf{let}\; v = @\, t_0\; e_2\;\mathbf{in}\; e_3]\!] \\
[\![\mathbf{let}\; v = @\, t_0\; t_1\;\mathbf{in}\; e_2]\!] & = & \mathbf{let}\; v = @\,[\![t_0]\!]\;[\![t_1]\!]\;\mathbf{in}\;[\![e_2]\!]
\end{array}
$$

Proposition 1 *This rewrite system is strongly normalizing and confluent.*

Proof: The rules do not overlap. The first, second, and fifth rules rewrite a term into another term in which only proper sub-parts are subjected to rewriting. Therefore the only trouble could come from the third and fourth rules.

Associate an offset to each **let** expression as follows. All offsets are initialized to zero. If a **let** expression occurs as immediate sub-term (function or argument) in the declaration part of another **let** expression, then increment its offset prior to any rewriting. Applying the third or the fourth rules decrements the offset. The other rules do not increment the counter. Therefore, since we consider finite terms, this rewriting system terminates. □

This rewrite system is also meaning-preserving because (1) all **let** variables were newly introduced in the first step, therefore there is no possibility of name clash; and (2) under call-by-value, sequentializing **let** expressions is meaning-preserving.

Going back to the examples of last section, the term

$$\begin{array}{l}\lambda\, g\,.\,\lambda\, x\,.\;\mathbf{let}\; v_0 = @\,(\mathbf{let}\; v_1 = @\, g\, x\;\mathbf{in}\; v_1)\;(\lambda\, z\,.\,z)\\ \qquad\quad\;\mathbf{in}\; v_0\end{array}$$

gets transformed into

$$\begin{array}{l}\lambda\, g\,.\,\lambda\, x\,.\;\mathbf{let}\; v_1 = @\, g\, x\\ \qquad\quad\;\mathbf{in}\;\mathbf{let}\; v_0 = @\, v_1\;(\lambda\, z\,.\,z)\\ \qquad\qquad\;\;\mathbf{in}\; v_0\end{array}$$

4 Introducing Continuations

We first open a parenthesis on naming and Church encoding, to motivate why and how continuations are introduced as λ-abstractions. Then we will present the third step of the CPS transformation.

4.1 Church encoding

Let us quote Wand for the following comprehensive description of Church encoding [15]:

> "A higher-order abstract syntax, also sometimes called a Church encoding, is a representation of the abstract syntax of a phrase as a λ-term in which the binding relationships in the source phrase are represented using the binding relationships in the λ-term. For example, a quantified formula $\forall x.A$ in first-order logic might be represented as $\forall(\lambda x.A)$. This shows how x is bound in A."

Church encoding makes it possible to avoid *special forms* (*i.e.*, BNF productions) in functional languages.

For example, rather than writing the rec special form

$$\mathbf{rec}\; f = e$$

to bind f recursively to e, one can apply a fixpoint operator to a λ-abstraction

$$\mathit{fix}\;(\lambda\, f\,.\,e)$$

for the same effect.

For another example, rather than writing the escape special form

$$\mathbf{escape}\; k\;\mathbf{in}\; e$$

to capture the current continuation, one can apply the *call/cc* operator to a λ-abstraction

$$\mathit{call/cc}\;(\lambda\, k\,.\,e)$$

for the same effect [2].

Going back to our CPS transformation, Church encoding suggests a very natural way to name the intermediate values yielded by applications.

4.2 Naming intermediate values, revisited

In a nutshell, we can get rid of the **let** special form

$$\mathbf{let}\ v = @\, t_0\, t_1\ \mathbf{in}\ e$$

using Church encoding by writing

$$\mathit{combine}(\lambda\, v\, .\, e)(@\, t_0\, t_1)$$

where *combine* is a new combinator.

We can define *combine* as the usual macro-expansion of a **let** expression into a β-redex:

$$\begin{array}{lcl} \mathit{combine} & : & [A \rightarrow B] \rightarrow A \rightarrow B \\ \mathit{combine} & = & \lambda\, \kappa\, .\, \lambda\, a\, .\, @\, \kappa\, a \end{array}$$

Alternatively, we can view $\lambda\, v\, .\, e$ as the continuation of $@\, t_0\, t_1$ and define *combine* as the application of its *second* argument to its *first*:

$$\mathit{combine} \;\; = \;\; \lambda\, \kappa\, .\, \lambda\, a\, .\, @\, a\, \kappa$$

This alternative definition of *combine* leads us to continuation-passing style. Correspondingly, we represent λ-abstractions $\lambda\, x\, .\, e$ by writing $\lambda\, x\, .\, (\lambda\, k\, .\, e)$ to account for the continuation. The base case of trivial terms t is handled by applying the current continuation $@\, k\, t$, where k is inductively inherited from the lexically enclosing declaration $\lambda\, k\, .\, e$. This is captured in the following rewrite system.

4.3 Introducing Continuations

At this stage, here is the BNF of λ-terms as they are produced by the second step (*cf.* Section 3).

$$\begin{array}{lcl} e & ::= & t \mid s \\ t & ::= & x \mid \lambda\, x\, .\, e \\ s & ::= & \mathbf{let}\ v = @\, t_0\, t_1\ \mathbf{in}\ e \end{array}$$

To the two sorts of variables — the original ones (x) and those naming intermediate values (v) — let us add a third sort k to denote continuations. The following transformation introduces continuations in these λ-terms, transforming a term e into $\lambda\, k\, .\, e'$ and introducing new identifiers k. To describe the transformation we use the following notation. We write

$$\vdash\ e \rightsquigarrow \lambda\, k\, .\, e'$$

to state that e is transformed into $\lambda\, k\, .\, e'$. This judgement uses the following two auxiliary judgements. We write

$$k\ \vdash\ e\ \Rightarrow\ e'$$

to state that within a context where k denotes the current continuation, e is transformed into e'. We write

$$\vdash\ t\ \rightarrow\ t'$$

to state that trivial expressions t are transformed into t'.

$$\frac{k \;\vdash\; e \;\Rightarrow\; e'}{\vdash\; e \;\rightsquigarrow\; \lambda k \,.\, e'}$$

$$k \;\vdash\; x \;\Rightarrow\; @\, k\, x$$

$$\frac{k' \;\vdash\; e \;\Rightarrow\; e'}{k \;\vdash\; \lambda x \,.\, e \;\Rightarrow\; @\, k\, (\lambda x \,.\, \lambda k' \,.\, e')}$$

$$\frac{\vdash\; t_0 \;\rightarrow\; t_0' \qquad \vdash\; t_1 \;\rightarrow\; t_1' \qquad k \;\vdash\; e \;\Rightarrow\; e'}{k \;\vdash\; \mathbf{let}\; v = @\, t_0\, t_1 \;\mathbf{in}\; e \;\Rightarrow\; @\, (@\, t_0'\, t_1')\, (\lambda v \,.\, e')}$$

$$\vdash\; x \;\rightarrow\; x$$

$$\frac{k \;\vdash\; e \;\Rightarrow\; e'}{\vdash\; \lambda x \,.\, e \;\rightarrow\; \lambda x \,.\, \lambda k \,.\, e'}$$

where k and k' are fresh variables.

Equivalently, the inherited attribute k can be expressed with the following two-level specification, as in Section 1.1. The term e is transformed into $\underline{\lambda} k.\overline{@}[\![e]\!]\, k$, where $[\![\ ]\!]$ is defined inductively as follows.

$$\begin{array}{rcl}
[\![x]\!] & = & \overline{\lambda} k.\underline{@} k\, x \\
[\![\lambda x \,.\, e]\!] & = & \overline{\lambda} k.\underline{@} k\, (\underline{\lambda} x.\underline{\lambda} k'.\overline{@}[\![e]\!]\, k') \\
[\![\mathbf{let}\; v = @\, t_0\, t_1 \;\mathbf{in}\; e]\!] & = & \overline{\lambda} k.\underline{@}(\underline{@}[t_0]\, [t_1])\, (\underline{\lambda} v.\overline{@}[\![e]\!]\, k) \\
[x] & = & x \\
[\lambda x \,.\, e] & = & \underline{\lambda} x.\underline{\lambda} k.\overline{@}[\![e]\!]\, k
\end{array}$$

Going back to the example of Section 3, the term

$$\begin{array}{l}
\lambda g \,.\, \lambda x \,.\; \mathbf{let}\; v_1 = @\, g\, x \\
\qquad\qquad \mathbf{in}\; \mathbf{let}\; v_0 = @\, v_1\, (\lambda z \,.\, z) \\
\qquad\qquad\quad \mathbf{in}\; v_0
\end{array}$$

gets transformed into

$$\begin{array}{l}
\lambda k \,.\, @\, k \\
\qquad \lambda g \,.\, \lambda k' \,.\, @\, k' \\
\qquad\qquad \lambda x \,.\, \lambda k'' \,.\, @\, (@\, g\, x) \\
\qquad\qquad\qquad \lambda v_1 \,.\, @\, (@\, v_1\, (\lambda z \,.\, \lambda k''' \,.\, @\, k'''\, z)) \\
\qquad\qquad\qquad\quad \lambda v_0 \,.\, @\, k''\, v_0
\end{array}$$

which is the usual CPS counterpart of the λ_v-term

$$\lambda g \,.\, \lambda x \,.\, @\, (@\, g\, x)\, (\lambda z \,.\, z)$$

as produced by Fischer & Plotkin's CPS transformer.

5 Fischer & Plotkin's sequentialization

At this point, one might wonder why Fischer & Plotkin's CPS transformer looks the way it does. This section justifies its appearance.

5.1 Compositionality

As can be noticed, the sequentialization of Section 3 is not compositional — we are using "compositional" as in denotational semantics.

Definition 1 *A rewrite system is* compositional *if the following holds for each equation: the part that is subjected to rewriting on the right hand side of the equation is a* proper subpart *of the left hand side of the equation.*

For example, the following equations of Section 3 are compositional

$$\begin{aligned} [\![x]\!] &= x \\ [\![\lambda x \,.\, e]\!] &= \lambda x \,.\, [\![e]\!] \\ [\![\mathbf{let}\ v = @\, t_0\, t_1\ \mathbf{in}\ e_2]\!] &= \mathbf{let}\ v = @\, [\![t_0]\!]\, [\![t_1]\!]\ \mathbf{in}\ [\![e_2]\!] \end{aligned}$$

but the following ones are not compositional.

$$\begin{aligned} [\![\mathbf{let}\ v = @\, (\mathbf{let}\ w = e_0\ \mathbf{in}\ e_1)\, e_2\ \mathbf{in}\ e_3]\!] &= [\![\mathbf{let}\ w = e_0\ \mathbf{in}\ \mathbf{let}\ v = @\, e_1\, e_2\ \mathbf{in}\ e_3]\!] \\ [\![\mathbf{let}\ v = @\, t_0\, (\mathbf{let}\ w = e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3]\!] &= [\![\mathbf{let}\ w = e_1\ \mathbf{in}\ \mathbf{let}\ v = @\, t_0\, e_2\ \mathbf{in}\ e_3]\!] \end{aligned}$$

Let us write a compositional sequentialization.

5.2 A compositional post-order traversal

The sequentialization of Section 3 is obtained by a post-order traversal of application nodes, in each λ-abstraction. There are many ways to program a post-order traversal. For example, a compositional traversal operating in linear time requires an accumulator. This accumulator can be represented as a function. Our point here (developed in Appendix A) is that *Fischer & Plotkin's CPS transformer precisely encodes a post-order traversal using a functional accumulator.*

Accordingly, the following rewrite system sequentializes **let** expressions compositionally using a functional accumulator. An expression e is sequentialized by

$$\overline{@}[\![e]\!]\, (\overline{\lambda} x.x)$$

where $[\![\ ]\!]$ is defined inductively as follows.

$$\begin{aligned} [\![x]\!] &= \overline{\lambda}\kappa.\overline{@}\kappa\, x \\ [\![\lambda x \,.\, e]\!] &= \overline{\lambda}\kappa.\overline{@}\kappa\, (\overline{@}[\![e]\!]\, (\overline{\lambda} x.x)) \\ [\![\mathbf{let}\ v = @\, e_0\, e_1\ \mathbf{in}\ v]\!] &= \overline{\lambda}\kappa.\overline{@}[\![e_0]\!]\, (\overline{\lambda} t_0.\overline{@}[\![e_1]\!]\, (\overline{\lambda} t_1.\underline{\mathbf{let}}\ v = \underline{@} t_0\, t_1\ \underline{\mathbf{in}}\ \overline{@}\kappa\, v)) \end{aligned}$$

This rewriting system corresponds to the SML function of Figure 7, in Appendix. In each abstraction, all applications are traversed in post-order. This traversal mimics the call-by-value reduction strategy.

5.3 Combining naming and sequentialization

In retrospect, isolating naming in a first phase was a pedagogic device. Let us compose the two first steps. Only the third line differs from the previous figure.

$$\begin{array}{rcl}
[\![x]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda\, x\, .\, e]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, (\overline{@}[\![e]\!]\, (\overline{\lambda}x.x)) \\
[\![@\, e_0\, e_1]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![e_0]\!]\, (\overline{\lambda}t_0.\overline{@}[\![e_1]\!]\, (\overline{\lambda}t_1.\underline{\mathbf{let}}\; v = \underline{@}t_0\, t_1\; \underline{\mathbf{in}}\; \overline{@}\kappa\, v))
\end{array}$$

v is a fresh variable.

6 The Essence of the CPS transformation

Specializing Fischer & Plotkin's equations to a restricted source syntax where the evaluation of all sub-expressions has already been sequentialized yields precisely the same equations as in Section 4.3.

Further, composing the transformations of Section 4.3 and Section 5.3 yields the specification of Section 1.1.

7 Comparison with Related Work

The CPS transformation is ubiquitous in many different areas of Computer Science, including logic, constructive mathematics, programming languages, and programming [9, 10]. The present approach focuses more on syntactic aspects of this transformation than on its semantic implications [7].

We have concentrated on producing output as a λ-term; other output formats may correspond to code generation and register allocation *à la* Wand. In particular the post-order traversal encoded in the CPS transformation justifies why the CPS transformation naturally supports semantics-directed compiling with respect to the abstract syntax tree and code "sequentialization" [1, 14, 15].

Our three steps for the CPS transformation also nicely generalize Haynes's CPS transformation [9], in that the post-order traversal of abstract syntax trees becomes explicit. This identification strenghtens the traditional intuitive descriptions and programming of the CPS transformation.

8 Conclusions and Issues

We have presented how to stage the CPS transformation into three steps, each of which is very simple to state and to motivate. This staging appeared sufficiently original and enlightening to warrant reporting it here, would it be only for pedagogic purposes. We have generalized it to extended and applied λ-calculi as offered *e.g.*, by Scheme [3, 4].

Our staging is general in that changing the traversal order of an abstract syntax tree yields another CPS transformation. Examples include right-to-left CPS_v-transformation and call-by-name — there one also needs to tag the source abstract syntax with **let**, **delay**, and **force**, as detailed in the full version of this paper.

Acknowledgements

To Karoline Malmkjær and Andrzej Filinski. Thanks are also due to John Greiner and Julia Lawall.

References

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

[2] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

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

[4] Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, Lecture Notes in Computer Science, Rennes, France, February 1992. (Technical report CIS-91-17, Department of Computing and Information Sciences, Kansas State University).

[5] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[6] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. Technical Report CIS-91-2, Kansas State University, Manhattan, Kansas, January 1991.

[7] Andrzej Filinski. Linear continuations. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, New Mexico, January 1992. ACM Press.

[8] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109. SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14, January 1972.

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

[10] Chetan R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Sixth Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991. IEEE.

[11] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

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

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

[14] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[15] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Steve Brooks, editor, *Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, March 1991. To appear.

A Compositional Tree Traversals

This appendix provides the necessary background about traversing trees compositionally. The point here is to show that Fischer & Plotkin's CPS transformer traverses syntax trees compositionally, using a functional accumulator. We are using SML because it makes the forms of source and target terms explicit and relates them to the term grammars.

A.1 Flattening trees

Let us define trees with zero-ary, unary, and binary nodes (*cf.* Figure 1). We want to flatten such trees into canonical form. For example, the tree

```
CONS(CONS(LEAF(80),
          NODE(CONS(CONS(LEAF(10),
                         LEAF(20)),
                    LEAF(30)))),
     LEAF(90))
```

is flattened into the following canonical form.

```
CONS(LEAF(80),
     CONS(NODE(CONS(LEAF(10),
                    CONS(LEAF(20),
                         LEAF(30)))),
          LEAF(90)))
```

Undergraduate data structures books describe how to flatten such a tree incrementally. This is captured in the iterative function displayed in Figure 2.

In the last line of its definition, `flatten` is passed a tree that is not a proper subpart of the input tree. In other terms, `flatten` *is not compositional with respect to its input* (see Section 5.1).

Is there a way to express `flatten` compositionally with respect to its input and yet making it traverse the tree only once? Undergraduate programming books describe how to do it — the idea is to use an accumulator to delay the recursive call on a proper sub-part of the input. This is captured in the function displayed in Figure 3.

Is there another way to express `flatten` compositionally with respect to its input? Graduate functional programming books describe how to do it — the idea is to represent the accumulator as a function. The datatype constructors are now functions:

```
fun NONE x = x;

fun SOME r l = CONS(l,r);
```

This is captured in the function displayed in Figure 4.

```
datatype 'a Tree = LEAF of 'a | NODE of 'a Tree | CONS of 'a Tree * 'a Tree
```

Figure 1: Specification of a tree in SML.

```
fun flatten (LEAF(a)) = LEAF(a)
  | flatten (NODE(t)) = NODE(flatten t)
  | flatten (CONS(LEAF(a),r)) = CONS(LEAF(a),flatten r)
  | flatten (CONS(NODE(t),r)) = CONS(NODE(flatten t),flatten r)
  | flatten (CONS(CONS(ll,lr),r)) = flatten(CONS(ll,CONS(lr,r)))
```

Figure 2: Flattening a tree in SML non-compositionally.

```
datatype 'a option = NONE | SOME of 'a
;
fun flatten t =
    let fun traverse (LEAF(a)) NONE = LEAF(a)
          | traverse (LEAF(a)) (SOME(z)) = CONS(LEAF(a),z)
          | traverse (NODE(t)) NONE = NODE(flatten t)
          | traverse (NODE(t)) (SOME(z)) = CONS(NODE(flatten t),z)
          | traverse (CONS(l,r)) acc = traverse l (SOME(traverse r acc))
    in traverse t NONE
    end
```

Figure 3: Flattening a tree in SML compositionally.

```
fun flatten t =
    let fun traverse (LEAF(a)) k = k (LEAF(a))
          | traverse (NODE(t)) k = k (NODE(flatten t))
          | traverse (CONS(l,r)) k = traverse l (fn z => CONS(z,traverse r k))
    in traverse t (fn x => x)
    end
```

Figure 4: Flattening a tree in SML compositionally with a functional accumulator.

A.2 Assessment

As demonstrated in the previous section, there is nothing inherently higher-order in flattening a tree based on a post-order traversal. In this section, we instantiate this result to λ-terms and we show that this instantiation coincides with Fischer & Plotkin's sequentialization (*cf.* Section 5).

The trees above closely resemble abstract syntax trees for the λ-calculus. Leaves correspond to variables; unary nodes correspond to λ-abstractions; binary nodes correspond to applications.

Let us go back to syntax trees where the intermediate values of applications are given a name (*cf.* Figure 5). Again, we distinguish between trivial terms (variables, abstractions) and serious terms (applications). Therefore, this data type should be seen as an implementation of the BNF specified in Section 2. For example, the term

$$\begin{array}{l}\lambda g\,.\,\lambda x\,.\,\mathbf{let}\ v_0 = @\,(\mathbf{let}\ v_1 = @\,g\,x\ \mathbf{in}\ v_1)\,(\lambda z\,.\,z)\\ \qquad\quad\ \mathbf{in}\ v_0\end{array}$$

is represented as

```
TV1(ABS1("g",
         TV1(ABS1("x",
                  SR1(APP1("v_0",
                           SR1(APP1("v_1",
                                    TV1(VAR1 "g"),
                                    TV1(VAR1 "x"))),
                           TV1(ABS1("z",
                                    TV1(VAR1 "z")))))))))
```

We want to sequentialize the intermediate applications into the data structure specified in Figure 6. This data type should be seen as an implementation of the BNF specified in Section 3. The example above is sequentialized into

```
TV2(ABS2("g",
         TV2(ABS2("x",
                  SR2(APP2("v_1",
                           VAR2 "g",
                           VAR2 "x",
                           SR2(APP2("v_0",
                                    VAR2 "v_1",
                                    ABS2("z",
                                         TV2(VAR2 "z")),
                                         TV2(VAR2 "v_0")))))))))
```

that corresponds to the term

$$\begin{array}{l}\lambda g\,.\,\lambda x\,.\,\mathbf{let}\ v_1 = @\,g\,x\\ \qquad\qquad\ \mathbf{in}\ \mathbf{let}\ v_0 = @\,v_1\,(\lambda z\,.\,z)\\ \qquad\qquad\qquad\ \mathbf{in}\ v_0\end{array}$$

This sequentialization can be achieved either non-compositionally or compositionally. Our point here is that the compositional sequentialization with a functional accumulator (*cf.* Figure 7) literally corresponds to Fischer & Plotkin's sequentialization (*cf.* Figure 8).

```
  datatype AST1 = TV1 of trivialAST1
                | SR1 of seriousAST1
and trivialAST1 = VAR1 of string
                | ABS1 of string * AST1
and seriousAST1 = APP1 of string * AST1 * AST1
```

Figure 5: Syntax of the λ-calculus in SML.

```
  datatype AST2 = TV2 of trivialAST2
                | SR2 of seriousAST2
and trivialAST2 = VAR2 of string
                | ABS2 of string * AST2
and seriousAST2 = APP2 of string * trivialAST2 * trivialAST2 * AST2
```

Figure 6: Syntax of the sequentialized λ-calculus in SML.

```
fun flatten ast =
    let fun traverse (TV1 t) k = traverse_t t k
          | traverse (SR1 s) k = traverse_s s k
        and traverse_t (VAR1 v) k = k (VAR2 v)
          | traverse_t (ABS1(v,ast)) k = k (ABS2(v,flatten ast))
        and traverse_s (APP1(v,fct,arg)) k
            = traverse fct (fn tfct
                           => traverse arg (fn targ
                                            => SR2(APP2(v,tfct,targ,k (VAR2 v)))))
    in traverse ast (fn t => (TV2 t))
    end
```

Figure 7: Sequentializing λ-calculus terms compositionally in SML.

$$
\begin{array}{rcl}
[\![x]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\, x \\
[\![\lambda\, x\, .\, e]\!] & = & \overline{\lambda}\kappa.\overline{@}\kappa\,(\overline{@}[\![e]\!]\,(\overline{\lambda}x.x)) \\
[\![\mathbf{let}\ v = @\, e_0\, e_1\ \mathbf{in}\ v]\!] & = & \overline{\lambda}\kappa.\overline{@}[\![e_0]\!]\,(\overline{\lambda}t_0.\overline{@}[\![e_1]\!]\,(\overline{\lambda}t_1.\underline{\mathbf{let}}\ v = \underline{@}t_0\, t_1\ \underline{\mathbf{in}}\ \overline{@}\kappa\, v))
\end{array}
$$

Figure 8: Fischer & Plotkin's sequentialization of λ-terms.