RICE UNIVERSITY

# The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms

by

**Amr A. Sabry**

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

**Doctor of Philosophy**

Approved, Thesis Committee:

Matthias Felleisen, Chairman
Professor of Computer Science

Robert Cartwright
Professor of Computer Science

Keith Cooper
Associate Professor of Computer Science

Steve Cox
Assistant Professor of Computational and Applied Mathematics

Houston, Texas

August, 1994

# Contents

# Chapter 2

# A History of CPS Transformations

We begin with some historical context motivating the development of CPS transformations. The first section presents our personal perspective on the discovery of the concept of CPS transformations. For a different and more extensive historical perspective, we refer the reader to Reynolds's survey [79]. In the second section, we apply the original CPS transformation (due to Fischer [37], Morris [69], and Strachey and Wadsworth [101]) to a pure call-by-value language. We also discuss Plotkin's analysis of the properties of this transformation [75]. Finally, we include a series of CPS transformations from the literature that represent various practical and theoretical improvements of the original formulation. The development culminates with our optimal compacting CPS transformation [83, 85].

## 2.1 History and Motivation: Procedure Call versus Goto

Unlike earlier languages, Algol 60 included both lexically-nested procedures (blocks) and unconditional jumps (gotos). The combination of these two constructs posed a challenge to both semanticists and implementors alike. On one hand, the contemporary mathematical theory of semantics [89, 90] could not accommodate imperative facilities such as gotos. On the other hand, the ability to jump outside of blocks (and hence jump to new lexical scopes) complicated the implementation of procedures and the representation of labels.

In an attempt to solve the above problems, two lines of research evolved. The first, attempting to solve the implementation problems, concentrated on transforming procedures calls to gotos. The second, attempting to solve the semantic problems, concentrated on transforming gotos to function calls. Surprisingly, both lines of research resulted in the *same* program transformation: the CPS transformation.

### 2.1.1 Procedure Call as Goto

Intuitively, procedure calls could be implemented as jumps (with arguments) if they were not required to return. In this case, each procedure (not the underlying implementation) would be responsible for the rest of the computation following its call. To realize this idea, a procedure must take an additional argument representing the rest of the computation, *i.e.*, a continuation.

The above idea can be traced back to A. van Wijngaarden. In the discussion following the presentation of his paper at an IFIP Working Conference on Formal Language Description Languages [96], van Wijngaarden argued that the presence of goto statements complicates the implementation of procedures and proposed a new strategy in which

> no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure return ...it's exactly the same as a goto, only called in other words [96, p 24].

Independently, Fischer used the same idea to implement languages with higher-order procedures such as Lisp, Scheme, and ML [37]. In such languages, a variable bound in an inner block (or procedure) may be referenced from outside. For example, in the expression:

```
((lambda (f) (f 1))
 (let ([y 2])
   (lambda (x) (+ y x))))
```

the variable $y$ is declared inside the boxed expression and is referenced during the evaluation of the expression $(f\ 1)$ outside the box. This property rules out a stack-based implementation strategy where the storage for $y$ is created when control enters the box and is discarded upon exit. Apparently, the semantics requires an implementation strategy that retains all bindings indefinitely in a heap-based garbage-collected store. The intuition of Fischer is that

> deletion takes place only when a function returns a value; if the function never returns (except at the end), no deletion takes place! Thus, ...instead of $g$ passing its result back to the caller, the caller is passed

> to $g$ as an additional argument. $g$ then applies the new argument to the result it used to return, thereby avoiding the necessity of returning immediately [37, p 107].

Based on this intuitive idea, Fischer proves that all programs can be mapped to a restricted subset of the original language that can be evaluated using a stack-based strategy [37].[1]

In summary, transforming program fragments so that they manipulate their continuation explicitly enables both an implementation of procedure calls as jumps and a stack-based implementation strategy of languages with higher-order procedures.

### 2.1.2 Goto as Procedure Call

The idea of defining the semantics of programming languages mathematically was proposed by Scott and Strachey [89, 90, 100]. Intuitively, the goal of the approach is to use well-established mathematical concepts, *e.g.*, functions, lattices, etc, in order to describe the semantics of programming languages.

In the absence of gotos and blocks, the semantics of an Algol-like language is defined by mapping each statement $S$ to a *state transformer* $\mathcal{T}[\![S]\!]$. The latter is a mathematical function mapping the store $\sigma$ before the execution of the statement $S$ to the store afterwards. The sequencing of statements thus naturally corresponds to function composition:

$$\mathcal{T}[\![S_1; S_2]\!]\sigma = \mathcal{T}[\![S_2]\!](\mathcal{T}[\![S_1]\!]\sigma).$$

Unfortunately, this simple scheme breaks down if statement $S_1$ includes a jump to an external label. To get around this problem, Strachey and Wadsworth [101] modify the state transformers to take an additional argument that represents the (state transformation specified by the) rest of the program, *i.e.*, the continuation. Given an initial continuation $\kappa$, the meaning of the composite statement $S_1; S_2$ is:

$$\mathcal{T}[\![S_1; S_2]\!]\sigma\kappa = \mathcal{T}[\![S_1]\!]\sigma\kappa' \qquad \text{where } \kappa'(\sigma') = \mathcal{T}[\![S_2]\!]\sigma'\kappa.$$

If statement $S_1$ includes a jump to label $L$, it can simply ignore its continuation $\kappa'$ and use the continuation function representing the label $L$ instead.

In summary, the explicit manipulation of the continuation enables a mathematical theory based on functions to explain both sequencing and jumps uniformly as function applications.

---

[1] In practice however, the resulting stack must be treated like a garbage-collected heap since it is never popped during the execution of the program [5, 37].

### 2.1.3 Continuations and Transfers of Control

Although the transformations described in the two previous subsections appear to have dual effects, they are in fact identical transformations. In each case, we transform a program (the source program or the mathematical meta-program) in a way that manipulates the continuation explicitly. In the first case, the continuation is used to perform procedure returns, and in the second case the continuation is used to perform sequencing and jumps.

Since the salient aspect of the above common program transformation is to pass continuations as additional arguments to other procedures or functions, it is called the "continuation-passing style" transformation. Informally, we can already verify that in a CPS program, all procedure calls can be implemented as gotos (Subsection 2.1.1) and all gotos can be implemented as procedure calls (Subsection 2.1.2). This ability to shift perspective from a high-level functional view to a low-level machine view and vice-versa is perhaps the most appealing aspect of CPS programs.[2]

## 2.2 The Original CPS Transformation

The original formulation of the CPS transformation is due to Fischer [37], Morris [69], and Strachey and Wadsworth [101]. We first illustrate the transformation of a pure (without constants) higher-order call-by-value language.

**Definition 2.1** *(Fischer CPS)* Let $k$, $m$, $n \in \mathit{Vars}$ be variables that do not occur in the argument to $\mathcal{F}$:

$$\begin{array}{rcl} \mathcal{F} : \Lambda & \rightarrow & \Lambda \\ \mathcal{F}[\![V]\!] & = & \lambda k.(k\ \mathcal{F}_v[\![V]\!]) \\ \mathcal{F}[\![MN]\!] & = & \lambda k.(\mathcal{F}[\![M]\!]\ (\lambda m.(\mathcal{F}[\![N]\!]\ \lambda n.((m\ k)\ n)))) \end{array}$$

$$\begin{array}{rcl} \mathcal{F}_v : \mathit{Values}(\Lambda) & \rightarrow & \Lambda \\ \mathcal{F}_v[\![x]\!] & = & x \\ \mathcal{F}_v[\![\lambda x.M]\!] & = & \lambda k.\lambda x.(\mathcal{F}[\![M]\!]\ k) \end{array}$$

[2]In fact, the CPS transformation encodes most common transfers of control, *e.g.*, sequencing, procedure calls, procedure returns, jumps, exceptions, coroutines, etc, using *one* mechanism.

In the transformed programs, the variable $k$ represents the continuation. As informally explained in Subsection 2.1.1, each procedure in the original program is modified to take the continuation as an additional argument, and each procedure return is replaced by an invocation of the continuation.

By inspection, the CPS transformation captures two important aspects of the semantics of the source language. First, the argument $N$ of an application $(M\ N)$ is evaluated before the procedure call, *i.e.*, the parameter-passing mechanism is call-by-value.[3] Second, the evaluation of the application $(M\ N)$ proceeds from left to right. It is trivial to rewrite the CPS transformation to encode a right-to-left evaluation order:

$$\mathcal{F}[\![MN]\!] = \lambda k.(\mathcal{F}[\![N]\!]\ (\lambda n.(\mathcal{F}[\![M]\!]\ \lambda m.((m\ k)\ n)))).$$

The main formal properties of the original CPS transformation were established by Fischer [37], Reynolds [77], and Plotkin [75]. The following theorem summarizes these main properties.

**Theorem 2.1** *(Plotkin [75])* Let $M \in \Lambda$.

**Simulation:** $\mathcal{F}_v[\![eval_v(M)]\!] = eval_v(\mathcal{F}[\![M]\!]\ (\lambda x.x))$
**Indifference:** $eval_n(\mathcal{F}[\![M]\!]\ (\lambda x.x)) = eval_v(\mathcal{F}[\![M]\!]\ (\lambda x.x))$
**Translation:** If $\boldsymbol{\lambda}\beta_v \vdash M = N$ then $\boldsymbol{\lambda}\beta \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$. The implication is not reversible. Also, $\boldsymbol{\lambda}\beta_v \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$ if and only if $\boldsymbol{\lambda}\beta \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$.

The simulation theorem shows that the evaluation of the CPS program produces correct outputs. The indifference theorem establishes that this evaluation yields the same result under call-by-value and call-by-name. The translation theorem establishes the soundness and *incompleteness* of $\boldsymbol{\lambda}\beta_v$ for reasoning about CPS programs.

## 2.3 Compacting CPS Programs

The main disadvantage of the Fischer CPS transformation is that it increases the size of source terms considerably. For example, we have:

$$\begin{aligned}\mathcal{F}[\![((\lambda x.x)\ y)]\!] \;=\; & \lambda k.((\lambda k.(k\ \lambda k.\lambda x.((\lambda k.kx)\ k))) \\ & \qquad (\lambda m.((\lambda k.ky) \\ & \qquad\qquad (\lambda n.((m\ k)\ n))))).\end{aligned}$$

[3] For other parameter-passing mechanisms, *e.g.*, call-by-name, call-by-need, we refer the reader to the literature [74, 75].

Not only is this explosion in size undesirable in practical settings, *e.g.*, when CPS programs are used as a compiler's intermediate representation, but it also complicates the relationship between source programs and their CPS counterparts. For example, the term $((\lambda x.x)\ y)$ reduces by one $\beta_v$-reduction to $y$. After CPS conversion, the reduction sequence becomes:

$$
\begin{array}{rcl}
\mathcal{F}[\![((\lambda x.x)\ y)]\!] & \longrightarrow & \lambda k.((\lambda m.((\lambda k.ky)\ (\lambda n.((m\ k)\ n)))) \\
& & \qquad (\lambda k.\lambda x.((\lambda k.kx)\ k))) \\
& \longrightarrow & \lambda k.((\lambda k.ky) \\
& & \qquad (\lambda n.(((\lambda k.\lambda x.((\lambda k.kx)\ k))\ k)\ n)) \\
& \longrightarrow & \lambda k.((\lambda n.(((\lambda k.\lambda x.((\lambda k.kx)\ k))\ k)\ n))\ y) \\
& \longrightarrow & \lambda k.(((\lambda k.\lambda x.((\lambda k.kx)\ k))\ k)\ y) \\
& \longrightarrow & \lambda k.((\lambda x.((\lambda k.kx)\ k))\ y) \\
(*) & \longrightarrow & \lambda k.((\lambda k.ky)\ k) \\
& \longrightarrow & \lambda k.(k\ y) \\
& = & \mathcal{F}[\![y]\!].
\end{array}
$$

In the sequence, only the reduction marked $(*)$ is significant as it corresponds to the source reduction. Following Plotkin's terminology [75], we call this reduction *proper*, and all other reductions *administrative*. The remainder of this chapter addresses various algorithms for the elimination of administrative redexes.

Steele [97] based his compiler for Scheme on the CPS transformation. In order to minimize the size of intermediate programs, Steele improved the original CPS transformation to recognize various special cases. Although Steele's CPS transform generates terms without administrative redexes, his algorithm is complicated and includes ad-hoc optimizations.

Danvy and Filinski [21] analyzed the CPS transformation in a more systematic way and defined a one-pass transformation that combines the traditional CPS transformation with the elimination of the administrative redexes. Their transformation is the result of a binding time analysis of the Fischer CPS transformation that identifies "static" applications that are then reduced *during* the translation:

$$
\begin{aligned}
\mathcal{D}[\![x]\!](k) &= k([\![x]\!]) \\
\mathcal{D}[\![\lambda x.M]\!](k) &= k([\![\lambda x.\lambda k.\mathcal{D}[\![M]\!](k_m)]\!]) \qquad \text{where } k_m(a) = (k\ a) \\
\mathcal{D}[\![MN]\!](k) &= \mathcal{D}[\![M]\!](k_m) \\
&\quad \text{where } k_m(a) = \mathcal{D}[\![N]\!](k_n) \\
&\qquad\quad \text{where } k_n(b) = ((a\ b)\ (\lambda a.k(a)))
\end{aligned}
$$

In the above transformation, the mathematical notation $f(a)$ refers to an application reduced at translation time; the syntax of the source (transformed) language remains unchanged.

A simple improvement of the Danvy/Filinski CPS transformation is possible if the order of the arguments to a procedure is switched so that the continuation becomes the first argument [85]. For example, according to the previous translation, the CPS transform of $(((\lambda x.\lambda y.x)\ a)\ b)$ is:

$$\lambda k.((\lambda x.\lambda k_1.(k_1\ \lambda y.\lambda k_2.k_2 x))\ a\ (\lambda m.(mb)k)).$$

Switching the order of arguments to the procedures yields:

$$\lambda k.((\lambda k_1.\lambda x.(k_1\ \lambda k_2.\lambda y.k_2 x))\ (\lambda m.(mk)b)\ a),$$

which has an administrative redex. The latter term can be reduced to a normal form without administrative redexes:

$$
\begin{aligned}
& \lambda k.(\lambda x.((\lambda m.(mk)b)\ \lambda k_2.\lambda y.k_2 x))\ a) \\
\longrightarrow\ & \lambda k.((\lambda x.((\lambda k_2.\lambda y.k_2 x)\ k)\ b)\ a) \\
\longrightarrow\ & \lambda k.((\lambda x.((\lambda y.kx)\ b))\ a).
\end{aligned}
$$

## 2.4 Sabry/Felleisen CPS Transformation $\mathcal{C}_k$

Although the Danvy/Filinski method eliminates administrative redexes from the output of the Fischer CPS transformation, the technique does not associate any semantic significance with administrative *reductions*. A thorough analysis of these reductions will lead us to an alternative compacting CPS transformation, which is crucial for the development in later chapters.

We begin with a precise definition of administrative reductions. To this end, we modify the Fischer CPS transformation by over-lining all $\lambda$-abstractions that are introduced during the translation. A reduction that involves any of these over-lined $\lambda$-abstractions constitutes an administrative reduction.

**Definition 2.2** *($\mathcal{F}$, $\overline{\beta}$, $\overline{\eta}$)* Let $k, m, n \in \mathit{Vars}$ be variables that do not occur in the argument to $\mathcal{F}$.

$$\begin{array}{rcl} \mathcal{F} : \Lambda & \rightarrow & \Lambda \\ \mathcal{F}[\![V]\!] & = & \overline{\lambda} k.(k\ \mathcal{F}_v[\![V]\!]) \\ \mathcal{F}[\![MN]\!] & = & \overline{\lambda} k.(\mathcal{F}[\![M]\!]\ (\overline{\lambda} m.(\mathcal{F}[\![N]\!]\ \overline{\lambda} n.((m\ k)\ n)))) \end{array}$$

$$\begin{array}{rcl} \mathcal{F}_v : \mathit{Values}(\Lambda) & \rightarrow & \Lambda \\ \mathcal{F}_v[\![x]\!] & = & x \\ \mathcal{F}_v[\![\lambda x.M]\!] & = & \overline{\lambda} k.\lambda x.(\mathcal{F}[\![M]\!]\ k) \end{array}$$

A $\beta$- or $\eta$-reduction is an administrative reduction if it involves over-lined abstractions:

$$\begin{array}{rcll} ((\overline{\lambda} x.M)\ N) & \longrightarrow & M[x := N] & & (\overline{\beta}) \\ (\overline{\lambda} x.Mx) & \longrightarrow & M & x \notin FV(M) & (\overline{\eta}) \end{array}$$

Given the modified Fischer CPS transformation, the construction of a CPS term without any administrative redexes can be formalized using the relation $\mathcal{F}2$ defined below.

**Definition 2.3** *($\mathcal{F}2$)* Let $M \in \Lambda$, then $\mathcal{F}2[\![M]\!] = P$ if and only if $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![M]\!] = P$ and $P$ is in $\overline{\beta}\overline{\eta}$-normal form.

The relation $\mathcal{F}2$ is a well-defined function: Each source term is related by $\mathcal{F}2$ to exactly one CPS term in $\overline{\beta}\overline{\eta}$-normal form.

**Proposition 2.2** The relation $\mathcal{F}2$ is a total function from $\Lambda$ to $\Lambda$.

**Proof** It is sufficient to show that $\mathcal{F}[\![M]\!]$ has a unique $\overline{\beta}\overline{\eta}$-normal form. By Lemma 2.4, $\overline{\beta}\overline{\eta}$-normal forms are unique. By Lemma 2.3, all reductions paths starting at $\mathcal{F}[\![M]\!]$ for $M \in \Lambda$ terminate. $\square$

**Lemma 2.3** Let $M \in \Lambda$. If $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![M]\!] \equiv M_0 \longrightarrow M_1 \longrightarrow M_2 \cdots$, then:

1. for all $M_i$, the bound variable of a $\overline{\lambda}$-abstraction occurs exactly once in the body,

2. for all $i \geq 0$, $M_{i+1}$ has one less $\overline{\lambda}$-abstraction than $M_i$, and
3. for some finite $n$, $M_n$ is in $\overline{\beta\eta}$-normal form.

**Proof** The first claim is true by construction, and is preserved by $\overline{\beta\eta}$-reductions. It implies that reductions cannot eliminate or duplicate sub-terms. Therefore the second claim holds. The last claim follows from the first two by induction on the number of $\overline{\lambda}$-abstractions in $\mathcal{F}[\![M]\!]$. □

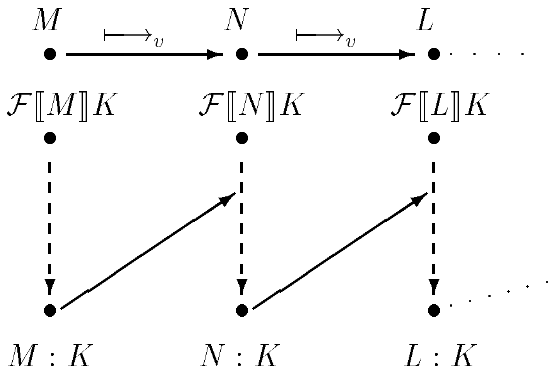
To complete the proof of Proposition 2.2, we show that if $\mathcal{F}[\![M]\!]$ reduces to two normal forms $P$ and $Q$, then $P$ and $Q$ are identical.

> **Lemma 2.4** Let $P$ and $Q$ be in $\overline{\beta\eta}$-normal form. If $\boldsymbol{\lambda}\overline{\beta\eta} \vdash \mathcal{F}[\![M]\!] \twoheadrightarrow P$ and $\boldsymbol{\lambda}\overline{\beta\eta} \vdash \mathcal{F}[\![M]\!] \twoheadrightarrow Q$, then $P \equiv Q$.

**Sketch** The proof is a consequence of the Church-Rosser theorem for $\beta\eta$ [6]. □

We can view the function $\mathcal{F}2$ as the specification of an optimal CPS transformation that eliminates all administrative reductions. It remains to find a one-pass implementation of this CPS transformation. By Proposition 2.2, administrative reductions can be performed in any order without affecting the result. Therefore, we can analyze a particular order, *e.g.*, the standard reduction sequence.

According to Plotkin [75], the standard reduction sequence of a source program relates to the standard reduction sequence of its CPS counterpart as described in the following diagram:




The term $M : K$ is the result of eliminating all the administrative reductions before the first (in a standard reduction sequence) proper reduction in $(\mathcal{F}[\![M]\!]\ K)$. The solid lines represent the reduction of source redexes; the dashed lines correspond to the reduction of administrative redexes.

From the above diagram, it is apparent that the role of the administrative reductions is to locate the next source redex in a standard reduction sequence, and to restructure the CPS program such that this redex occurs at the top level. For example, in the term:

$$M \stackrel{df}{=} ((\lambda y.y)\ ((\lambda x.x)\ z)),$$

the first redex in a standard reduction sequence is $((\lambda x.x)\ z)$. After CPS conversion and the elimination of all administrative reductions, we get:

$$P \stackrel{df}{=} (\mathcal{F}2[\![M]\!]\ (\lambda a.a)) = ((\lambda x.((\lambda y.((\lambda a.a)\ y))\ x))\ z),$$

where the redex $((\lambda x. \cdots)\ z)$ occurs at the top level.

Given that administrative reductions "lift" the next redex of a standard reduction sequence to the top-level, we can derive a compacting CPS transformation that eliminates all administrative redexes on the fly. The transformation relies on the fact that the first redex of a standard reduction sequence always occurs inside an evaluation context.

**Definition 2.4** *($\mathcal{C}_k, \Phi, \mathcal{K}_k$)* The CPS transformation uses three mutually recursive functions: $\mathcal{C}_k$ to transform terms, $\Phi$ to transform values, and $\mathcal{K}_k$ to transform evaluation contexts. Let $k, u_i \in \mathit{Vars}$ be variables that do not occur in the argument to $\mathcal{C}_k$.[4]

$$\begin{array}{rcl}
\mathcal{C}_k : \Lambda & \rightarrow & \Lambda \\
\mathcal{C}_k[\![V]\!] & = & (k\ \Phi[\![V]\!]) \\
\mathcal{C}_k[\![E[(x\ V)]]\!] & = & ((x\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V]\!]) \\
\mathcal{C}_k[\![E[((\lambda x.M)\ V)]]\!] & = & ((\lambda x.\mathcal{C}_k[\![E[M]]\!])\ \Phi[\![V]\!]) \\
\\
\Phi : \mathit{Values}(\Lambda) & \rightarrow & \Lambda \\
\Phi[\![x]\!] & = & x \\
\Phi[\![\lambda x.M]\!] & = & \overline{\lambda} k.\lambda x.\mathcal{C}_k[\![M]\!]
\end{array}$$

[4]The CPS transformation $\mathcal{C}_k$ is, in spirit, similar to the CPS transformation by Friedman, Wand, and Haynes [41, ch 8], but differs significantly in its formal part.

$$\begin{array}{rcl} \mathcal{K}_k : EvCont(\Lambda) & \rightarrow & \Lambda \\ \mathcal{K}_k[\![ [\,] ]\!] & = & k \\ \mathcal{K}_k[\![E[(x\ [\,])]]\!] & = & (x\ \mathcal{K}_k[\![E]\!]) \\ \mathcal{K}_k[\![E[((\lambda x.M)\ [\,])]]\!] & = & (\lambda x.\mathcal{C}_k[\![E[M]]\!]) \\ \mathcal{K}_k[\![E[([\,]\ M)]]\!] & = & (\overline{\lambda} u_i.\mathcal{C}_k[\![E[(u_i\ M)]]\!]) \end{array}$$

The transformation of a complete program $M$ is $((\lambda k.\mathcal{C}_k[\![M]\!])\ (\lambda a.a))$. The function $\mathcal{C}_k$ is parameterized over a variable $k$ that represents the current continuation. The CPS transform of values is straightforward. The translation of $E[(x\ V)]$ generates a term in which the unknown procedure $x$ is applied to the continuation $\mathcal{K}_k[\![E]\!]$ and the result applied to the argument $\Phi[\![V]\!]$. The CPS transform of $E[((\lambda x.M)\ V)]$ conceptually lifts the redex outside the evaluation context producing $((\lambda x.E[M])\ V)$ and then converts the resulting term to CPS. The first three cases in the translation of evaluation contexts to continuations have the same intuitive explanation. In the last case $E[([\ ]\ M)]$, the term in function position is the result of an intermediate computation. The CPS transformation gives the intermediate result a fresh name $u_i$ and proceeds with the translation of a simpler term.

Although $\mathcal{C}_k$, $\Phi$ and $\mathcal{K}_k$ are not defined by structural induction, it is relatively easy to check that the functions are well-defined using an appropriate notion of "size".

> **Definition 2.5** *(Size)* The *size* of a term $M$ (denoted by $|M|$) is the number of variables in $M$ (including binding occurrences). The *size* of a context $E$ (denoted by $|E|$) is the number of variables in $E$ (including binding occurrences) plus 2.

In particular, the size of $E[(u_i\ M)]$ is smaller than the size of $E[([\ ]\ M)]$ because the empty context always replaces an application. Also, the size of $(\lambda x.M)$ is greater than the size of $M$ by 1.

The following proposition verifies that the outputs of $\mathcal{C}_k$ and $\mathcal{F}2$ are identical. As a consequence, the proof also establishes that $\mathcal{C}_k$ is a total function.

> **Proposition 2.5** Let $M \in \Lambda$. Then, $\mathcal{F}2[\![M]\!] \equiv \lambda k.\mathcal{C}_k[\![M]\!]$.

**Proof** By the definition of $\mathcal{F}2$, it suffices to establish the following statements:

- $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ k) = \mathcal{C}_k[\![M]\!]$.

- $\mathcal{C}_k[\![M]\!]$ is in $\overline{\beta\eta}$-normal form.

- $(\mathcal{F}[\![M]\!]\ k)$ has a unique $\overline{\beta\eta}$-normal form.

The last claim follows from Proposition 2.2. The proof of the first two claims is by induction on the size of the argument to $\mathcal{C}_k$. Since the transformation of a term by $\mathcal{C}_k$ refers to evaluation contexts, we extend the Fischer CPS transformation to accept evaluation contexts and strengthen the inductive hypothesis to take evaluation contexts into account.

The extension of the Fischer CPS transformation is the following:

$$\begin{array}{rcl} \mathcal{F} : \mathit{EvCont}(\Lambda) & \rightarrow & \Lambda \\ \mathcal{F}[\![[\ ]]\!] & = & \overline{\lambda} k.k \\ \mathcal{F}[\![(V\ E)]\!] & = & \overline{\lambda} k.(\mathcal{F}[\![V]\!]\ \overline{\lambda} m.(\mathcal{F}[\![E]\!]\ \overline{\lambda} n.((m\ k)\ n))) \\ \mathcal{F}[\![(E\ M)]\!] & = & \overline{\lambda} k.(\mathcal{F}[\![E]\!]\ \overline{\lambda} m.(\mathcal{F}[\![M]\!]\ \overline{\lambda} n.((m\ k)\ n))) \end{array}$$

The extended function satisfies the following properties which we state without proof:

- $\boldsymbol{\lambda}\overline{\beta} \vdash (\mathcal{F}[\![E[M]]\!]\ K) = (\mathcal{F}[\![M]\!]\ (\mathcal{F}[\![E]\!]\ K))$

- $\boldsymbol{\lambda}\overline{\beta} \vdash (\mathcal{F}[\![E[E']]\!]\ K) = (\mathcal{F}[\![E']\!]\ (\mathcal{F}[\![E]\!]\ K))$.

We can now prove the following statements by induction on the size of $G$, where $G$ is the argument to $\mathcal{C}_k$ or $\mathcal{K}_k$:

- $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![M]\!]\ k) = \mathcal{C}_k[\![M]\!]$ and $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![E]\!]\ k) = \mathcal{K}_k[\![E]\!]$;

- $\mathcal{C}_k[\![M]\!]$ and $\mathcal{K}_k[\![E]\!]$ are in $\overline{\beta\eta}$-normal form.

The proof proceeds by case analysis on the possible inputs to the functions $\mathcal{C}_k$ or $\mathcal{K}_k$:

1. $G = V$: then, $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = ((\overline{\lambda} k.k\mathcal{F}_v[\![V]\!])\ k) = (k\ \mathcal{F}_v[\![V]\!])$. By cases:

   (a) $V = x$: then $(k\ \mathcal{F}_v[\![V]\!]) \equiv (k\ x) \equiv \mathcal{C}_k[\![x]\!]$. Moreover, $\mathcal{C}_k[\![x]\!]$ is in $\overline{\beta\eta}$-normal form.

   (b) $V = \lambda x.M$: then $(k\ \mathcal{F}_v[\![V]\!]) \equiv (k\ \overline{\lambda} c.\lambda x.\mathcal{F}[\![M]\!]c)$. By the inductive hypothesis $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![M]\!]\ c) = \mathcal{C}_c[\![M]\!]$, and $\mathcal{C}_c[\![M]\!]$ is in $\overline{\beta\eta}$-normal form. The result follows since $\mathcal{C}_k[\![V]\!] \equiv (k\ \overline{\lambda} c.\lambda x.\mathcal{C}_k[\![M]\!])$.

2. $G = E[(x\ V)]$: then,

$$\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = (\mathcal{F}[\![(x\ V)]\!]\ (\mathcal{F}[\![E]\!]\ k)) = ((x\ \mathcal{F}[\![E]\!]k)\ \mathcal{F}_v[\![V]\!]).$$

There are two cases:

(a) $V \notin \mathit{Vars}$: then $|E| < |G|$. By the inductive hypothesis, $\boldsymbol{\lambda}\overline{\beta\eta} \vdash \mathcal{F}[\![E]\!]k = \mathcal{K}_k[\![E]\!]$, and $\mathcal{K}_k[\![E]\!]$ is in $\overline{\beta\eta}$-normal form. By an argument similar to case 1, $\boldsymbol{\lambda}\overline{\beta\eta} \vdash \mathcal{F}_v[\![V]\!] = \Phi[\![V]\!]$, and $\Phi[\![V]\!]$ is in $\overline{\beta\eta}$-normal form. The result follows since $\mathcal{C}_k[\![G]\!] \equiv ((x\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V]\!])$.

(b) $V \in \mathit{Vars}$: then $|E| = |G|$ and the inductive hypothesis does not apply. By in-lining the arguments in cases 4 to 7, $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![E]\!]\ k) = \mathcal{K}_k[\![E]\!]$, and $\mathcal{K}_k[\![E]\!]$ is in $\overline{\beta\eta}$-normal form. The result follows as in sub-case (a).

3. $G = E[((\lambda x.M)\ V)]$: then,

$$\begin{aligned}\boldsymbol{\lambda}\overline{\beta\eta} \vdash \mathcal{F}[\![G]\!]k &= (\mathcal{F}[\![((\lambda x.M)\ V)]\!]\ (\mathcal{F}[\![E]\!]k))\\ &= ((\lambda x.\mathcal{F}[\![E[M]]\!]k)\ \mathcal{F}_v[\![V]\!]).\end{aligned}$$

The result follows by the inductive hypothesis and an argument similar to case 1.

4. $G = [\ ]$: then $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = ((\overline{\lambda}k.k)\ k) = k = \mathcal{K}_k[\![G]\!]$. Moreover $\mathcal{K}_k[\![G]\!]$ is in $\overline{\beta\eta}$-normal form.

5. $G = E[(x\ [\ ])]$: then,

$$\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = (\mathcal{F}[\![(x\ [\ ])]\!]\ (\mathcal{F}[\![E]\!]k)) = (x\ \mathcal{F}[\![E]\!]k).$$

The result follows by the inductive hypothesis.

6. $G = E[((\lambda x.M)\ [\ ])]$: then $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = (\lambda x.\mathcal{F}[\![E[M]]\!]k)$, and the result follows by the inductive hypothesis.

7. $G = E[([\ ]\ M)]$: then,

$$\begin{aligned}\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) &= (\mathcal{F}[\![([\ ]\ M)]\!]\ (\mathcal{F}[\![E]\!]k))\\ &= (\overline{\lambda}u.\mathcal{F}[\![M]\!]\ (u\ (\mathcal{F}[\![E]\!]k)))\\ &= (\overline{\lambda}u.(\mathcal{F}[\![E[(u\ M)]]\!]\ k)).\end{aligned}$$

The size of $E[(u\ M)]$ is smaller than the size of $E[([\ ]\ M)]$ by 1. The inductive hypothesis implies $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![E[(u\ M)]]\!]\ k) = \mathcal{C}_k[\![E[(u\ M)]]\!]$, and $\mathcal{C}_k[\![E[(u\ M)]]\!]$

is in $\overline{\beta}\overline{\eta}$-normal form. Therefore, $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = \mathcal{K}_k[\![G]\!]$. By a simple case analysis, $\mathcal{C}_k[\![E[(u\ M)]]\!]$ is never of the form $(K\ u)$ for some term $K$. Therefore, no new $\overline{\eta}$-redex is created in $\overline{\lambda}u.\mathcal{C}_k[\![E[(u\ M)]]\!]$ and the term is in $\overline{\beta}\overline{\eta}$-normal form.

$\square$

## 2.5 Summary

The transformation to CPS unifies all transfers of control using procedure calls or jumps. In the process of transforming programs, the original algorithm increases the size of source terms considerably by introducing administrative redexes. Both Steele [97] and Danvy/Filinski [21] developed algorithms to eliminate administrative redexes but did not give any semantic significance to administrative reductions. We exploited the connection between administrative reductions and the standard reduction sequence to develop a new optimal CPS transformation.