LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993

Reasoning about Programs in Continuation-Passing Style*

AMR SABRY† (*sabry@cs.rice.edu*)

MATTHIAS FELLEISEN† (*matthias@cs.rice.edu*)

Department of Computer Science, Rice University, Houston, TX 77251-1892

Keywords: λ-calculus, λ_v-calculus, continuation-passing style, CPS transformations, inverse CPS transformations, λ_v-C-calculus, IOCC

Abstract. Plotkin's λ_v-calculus for call-by-value programs is weaker than the $\lambda\beta\eta$-calculus for the same programs in continuation-passing style (CPS). To identify the call-by-value axioms that correspond to $\beta\eta$ on CPS terms, we define a new CPS transformation and an inverse mapping, both of which are interesting in their own right. Using the new CPS transformation, we determine the precise language of CPS terms closed under $\beta\eta$-transformations, as well as the call-by-value axioms that correspond to the so-called administrative $\beta\eta$-reductions on CPS terms. Using the inverse mapping, we map the remaining β and η equalities on CPS terms to axioms on call-by-value terms. On the pure (constant free) set of Λ-terms, the resulting set of axioms is equivalent to Moggi's computational λ-calculus. If the call-by-value language includes the control operators *abort* and *call-with-current-continuation*, the axioms are equivalent to an extension of Felleisen *et al.*'s λ_v-C-calculus and to the equational subtheory of Talcott's logic IOCC.

Contents

*This article is a revised and extended version of the conference paper with the same title [42]. The technical report of the same title contains additional material.

†The authors were supported in part by NSF grant CCR 89-17022 and by Texas ATP grant 91-003604014.

List of Definitions

List of Figures

1. Compiling with and without Continuations

Many compilers for higher-order applicative languages use the continuation-passing style (CPS) transformation [21, 41] to generate an intermediate representation [3, 30, 43, 46]. The CPS intermediate language has the following desirable properties:

- The language consists only of basic primitive operations and procedure applications whose semantics is independent of the parameter evaluation technique [39, 41]; many optimizations are therefore sequences of β- and η-reductions [2, 43, 46].
- It exposes the control flow of the program; complicated control facilities in the source language, *e.g.*, exception handlers and *call-with-current-continuation* [40], are translated to simple procedures that manipulate their continuation arguments in non-standard ways [41, 47].
- It constitutes an abstract assembly language whose standard reduction sequence mimics the behavior of typical target machines [2, 3, 24, 29, 49].

The CPS transformation is a global transformation that affects every subexpression in a program. It restructures programs to the extent that many of their original aspects are unrecognizable. The transformation might even obscure the analysis of optimizations that rely on execution paths having matching call/return pairs [Private Communication, Hans Boehm, October 1992]. For these reasons and others, a fair number of compilers (usually called direct compilers) do not rely on the CPS intermediate representation [5, 6, 28, 32].

The choice of compilation strategy would not be significant if both classes of compilers performed the same optimizations. However for the important class of optimizations that is expressible in the framework of the λ-calculus, CPS compilers have an advantage. While CPS compilers can use the full λ-calculus reductions to perform optimizations, direct compilers can apparently only rely on weaker calculi like the λ_v-calculus [39] and the λ_v-C-calculus [18, 19]. Naturally we ask: what optimizations do CPS compilers perform that are not also performed by direct compilers? Or in technical terms, what set of call-by-value axioms *corresponds* to $\beta\eta$-reductions on CPS terms. With such a set, CPS compilers could report optimizations in terms of the original program, and direct compilers could benefit by performing all the optimizations that correspond to $\beta\eta$-reductions on CPS programs.

The main technical result of this paper is the identification of axioms for call-by-value languages that correspond to $\beta\eta\delta$-reductions on CPS programs.[1] In order to simplify the technical exposition, we first focus on a pure, constant-free call-by-value language. Specifically, we strengthen Plotkin's Translation Theorem (cf. Theorem 1) by identifying an axiom set X such that:

$$\boldsymbol{\lambda}\beta_v X \vdash M = N \qquad \textit{if and only if} \qquad \boldsymbol{\lambda}\beta\eta \vdash \mathit{cps}(M) = \mathit{cps}(N).$$

Our strategy for the identification of the axioms X consists of three steps:

1. First, we develop a new CPS transformation that produces a canonical form of CPS programs. The new transformation provides a simple characterization of a CPS language that is closed under $\beta\eta$-reductions. It also identifies a subset A of the call-by-value axioms X.

2. Second, we develop an "inverse" transformation that maps canonical CPS programs back to the original language. As Danvy and Lawall [11, 14] convincingly argue, this translation from CPS to direct terms is useful in its own right.

3. Finally, by studying the connection between the CPS transformation and the inverse mapping, we systematically derive the remaining axioms in the set X. The resulting calculus is a variant of Moggi's untyped computational λ-calculus [36].

A complete treatment of the relationship between programs and their CPS transforms must also analyze a language with control operators since these are the language facilities for which continuations were conceived [34, 37, 41, 47]. To this end, we extend our language with two typical control operators expressible in the CPS framework: *abort* and *call-with-current-continuation*. These operators suffice to express a wide variety of control abstractions such as error exits, jumps, intelligent backtracking, coroutines, and exception handling [23, 27].[2]

[1] For call-by-name languages, the reductions $\beta\eta$ are valid in both the source language and the CPS language and the CPS translation does not create any new opportunities for equational reasoning [39, page 153].

[2] The CPS framework can also express a control delimiting facility [44], *e.g.*, *prompt* [15]. However, we do not include *prompt* in our language for two reasons. First, current language implementations do not include such facilities. Second, the CPS translation of *prompt* generates expressions that are no longer independent of the timing of parameter evaluation [12, 20]. As a consequence, we can associate two different semantics with the CPS language: a call-by-value semantics which is the classic semantics for control delimiters [13, 15], and a call-by-name semantics which yields a *lazy prompt*. The first semantics is incompatible with the $\beta\eta$-reductions we use for the CPS language and, at this point, there seems to be little practical motivation to pursue the analysis of the lazy prompt.

The addition of *abort* and *call-with-current-continuation* extends the set of CPS programs and generates a new set of program equivalences that are provable using β- and η-reductions. In order to re-establish the correspondence between the source and CPS calculi, we extend both the CPS transformation and its inverse and proceed in the same way as for the pure language. The resulting calculus includes the reductions of the λ_v-C-calculus [18, 19] and is equivalent to the equational subtheory of the logic IOCC (Impredicative theory of Operations, Control abstractions, and Classes) [48].

Finally, since Lisp and similar languages include more facilities than just procedures and control operators, we consider the language Core Scheme, which also includes constants, conditionals, and assignments. We develop a theory for reasoning about Core Scheme that proves all the equations that hold in the CPS framework. To illustrate the power of the theory, we optimize a Scheme program that implements a coroutine facility using first-class continuations. The left program in Figure 1, due to Haynes,

```
(define make-coroutine
  (lambda (f)
    (callcc
      (lambda (maker)
         (let ([LCS 'any])
           (let ([resume
                    (lambda (dest val)
                      (callcc
                        (lambda (k)
                          (set! LCS k)
                          (dest val))))])
             (f resume
               (resume maker
                  (lambda (v) (LCS v))))
             (error "fell off end")))))))
```

```
(define make-coroutine
  (lambda (f)
    (letrec ([LCS
               (lambda (x)
                 (f (lambda (dest val)
                      (callcc
                        (lambda (k)
                          (set! LCS k)
                          (dest val))))
                    x)
                 (error "fell off end"))])
      (lambda (v) (LCS v)))))
```

Figure 1: Coroutines from Continuations.

Friedman, and Wand [27], is the result of clear and well-understood design steps but is far more complicated than necessary. The initialization part of the coroutine is non-trivial: it involves capturing a continuation and an artificial use of the procedure *resume* that assigns the proper value to the local control state (*LCS*). The second program, a variant of the first according to folklore, avoids the clumsy initialization phase and immediately returns a closure. Using our new set of axioms, we can rewrite the left program to the second. The derivation is the subject of Section 10.

The next section introduces the basic terminology and notation of the λ-calculus and its semantics. Sections 3 to 6 are dedicated to the pure call-by-value language. Section 3 includes a short history of CPS transformations that motivates the ideas that lead to our new CPS transformation in Section 4. The inverse mapping is the subject of Section 5. In Section 6, we identify the complete set of axioms X and prove its completeness with respect to β- and η-reductions on CPS terms. Section 7 extends the result to a language that includes the non-local control operators *abort* and *call-with-current-continuation*. Section 8 deals with the addition of constants, assignments, and conditionals. Finally, we conclude with a brief discussion of the theoretical and practical implications of our work and future directions of research. The appendix includes the tedious but straightforward portions of the proofs.

2. Λ: Calculi and Semantics

The language Λ is a pure (constant-free) functional language. The set of terms is generated inductively over an infinite set of variables *Vars*; it includes values and applications. Values consist of variables and λ-abstractions, applications are juxtapositions of terms:

$$\begin{array}{rcll} M & ::= & V \mid (M\ M) & (\Lambda) \\ V & ::= & x \mid (\lambda x.M) & (\mathit{Values}) \\ x & \in & \mathit{Vars} & \end{array}$$

We adopt Barendregt's [4, chapters 2, 3] notation and terminology for Λ's syntax. Thus, in the abstraction $(\lambda x.M)$, the variable x is *bound* in M. Variables that are not bound by a λ-abstraction are *free*; the set of free variables in a term M is $FV(M)$. A term is *closed* if it has no free variables. We identify terms modulo bound variables, and we assume that free and bound variables do not interfere in definitions or theorems. In short, we follow common practice and work with the quotient of Λ under α-equivalence. We write $M \equiv N$ for α-equivalent terms M and N.

The term $M[x := N]$ is the result of the capture-free substitution of all free occurrences of x in M by N, *e.g.*, $(\lambda x.xz)[z := (\lambda y.x)] \equiv (\lambda u.u(\lambda y.x))$. A *context* C is a term with a "hole", $[\ ]$, in the place of one subterm. The operation of *filling* the context C with a term M yields the term $C[M]$, possibly capturing some free variables of M in the process, *e.g.*, the result of filling $(\lambda x.x[\ ])$ with $(\lambda y.x)$ is $(\lambda x.x(\lambda y.x))$. The variables that may be captured when filling a context C are called the *trapped* variables of C and are denoted by $\mathit{trap}(C)$.

Calculi

A λ-calculus is an equational theory over Λ with a finite number of axiom schemas and inference rules. The most familiar axiom schemas, also called notions of reduction, are the following:

$$
\begin{array}{rcllr}
((\lambda x.M)\ N) & \longrightarrow & M[x := N] & N : \text{arbitrary} & (\beta) \\
((\lambda x.M)\ V) & \longrightarrow & M[x := V] & V : \text{Value} & (\beta_v) \\
\lambda x.Mx & \longrightarrow & M & x \notin FV(M) & (\eta) \\
\lambda x.Vx & \longrightarrow & V & x \notin FV(V) & (\eta_v)
\end{array}
$$

A reduction may also be applied to a context C_1 yielding another context C_2. In that case, the holes in both contexts are treated as placeholders to an arbitrary term. We only use such reductions when the sets of trapped variables in both contexts are empty (cf. evaluation contexts below).

The set of inference rules is identical for all λ-calculi. It extends some notions of reduction to an equivalence relation compatible with syntactic contexts:

$$
\begin{array}{rcllr}
M \longrightarrow N & \Rightarrow & C[M] = C[N] \text{ for all contexts } C & & (\mathit{Compatibility}) \\
 & & M = M & & (\mathit{Reflexivity}) \\
M = L, L = N & \Rightarrow & M = N & & (\mathit{Transitivity}) \\
M = N & \Rightarrow & N = M & & (\mathit{Symmetry})
\end{array}
$$

The underlying set of axioms completely identifies a theory. For example, β generates the theory $\boldsymbol{\lambda}\beta$, β_v generates the theory $\boldsymbol{\lambda}\beta_v$, and the union of β and η generates the theory $\boldsymbol{\lambda}\beta\eta$. In general, we write $\boldsymbol{\lambda}X$ to refer to the theory generated by a set of axioms X. When a theory $\boldsymbol{\lambda}X$ proves an equation $M = N$, we write $\boldsymbol{\lambda}X \vdash M = N$. If the proof does not use the inference rule (*Symmetry*), we write $\boldsymbol{\lambda}X \vdash M \longrightarrow\!\!\!\!\!\rightarrow N$.

A notion of reduction R is Church-Rosser (CR) if $\boldsymbol{\lambda}R \vdash M = N$ implies that there exists a term L such that both M and N reduce to L, *i.e.*, $\boldsymbol{\lambda}R \vdash M \longrightarrow\!\!\!\!\!\rightarrow L$ and $\boldsymbol{\lambda}R \vdash N \longrightarrow\!\!\!\!\!\rightarrow L$. A term M is in R-normal form if there are no R-reductions starting with M.

Many of the results in the paper relate the calculi for different languages. To provide a uniform terminology for the relationships between such systems, we introduce the concept of "equational correspondence".

Definition 1 (Equational Correspondence) *Let $\mathcal{S}$ and $\mathcal{T}$ be two languages with calculi $\boldsymbol{\lambda}X_{\mathcal{S}}$ and $\boldsymbol{\lambda}X_{\mathcal{T}}$ respectively. Also let $f : \mathcal{S} \rightarrow \mathcal{T}$ be a translation from $\mathcal{S}$ to $\mathcal{T}$, and $g : \mathcal{T} \rightarrow \mathcal{S}$ be a translation from $\mathcal{T}$ to $\mathcal{S}$. Finally let $s, s_1, s_2 \in \mathcal{S}$ and $t, t_1, t_2 \in \mathcal{T}$. Then the calculus $\boldsymbol{\lambda}X_{\mathcal{S}}$* equationally corresponds *to the calculus $\boldsymbol{\lambda}X_{\mathcal{T}}$ if the following four conditions hold:*

1. $\boldsymbol{\lambda} X_{\mathcal{S}} \vdash s = (g \circ f)(s)$.
2. $\boldsymbol{\lambda} X_{\mathcal{T}} \vdash t = (f \circ g)(t)$.
3. $\boldsymbol{\lambda} X_{\mathcal{S}} \vdash s_1 = s_2$ *if and only if* $\boldsymbol{\lambda} X_{\mathcal{T}} \vdash f(s_1) = f(s_2)$.
4. $\boldsymbol{\lambda} X_{\mathcal{T}} \vdash t_1 = t_2$ *if and only if* $\boldsymbol{\lambda} X_{\mathcal{S}} \vdash g(t_1) = g(t_2)$.

The above correspondence is similar to the correspondence between the λ-calculus and combinatory logic [4, 10]. In the third clause, the left-to-right implication refers to the *soundness* of the calculus $\boldsymbol{\lambda} X_{\mathcal{S}}$, and the right-to-left implication refers the *completeness* of the calculus $\boldsymbol{\lambda} X_{\mathcal{S}}$ (relative to $\boldsymbol{\lambda} X_{\mathcal{T}}$, f, and g).

Semantics

The semantics of the language Λ is a (partial) function, *eval*, from programs to answers. A *program* is a term with no free variables and, in practical languages, an *answer* is a member of the syntactic category of values. Typically, *eval* is defined via an abstract machine that manipulates abstract counterparts to hardware stacks, stores, registers, etc. Examples are the SECD machine [31] and the CEK machine [16].

An equivalent method for specifying the semantics is based on the Curry-Feys Standard Reduction Theorem [16, 39]. The Standard Reduction Theorem defines a partial function, $\longmapsto$, from programs to programs that corresponds to a single evaluation step of an abstract machine for Λ.

A standard step (i) decomposes the program into a special context E and a leftmost-outermost redex R (not inside an abstraction), and (ii) fills E with the contractum of R. The special contexts are *evaluation contexts* and have the following definition for the call-by-value and call-by-name variants of Λ, respectively [16]:

$$\begin{array}{rcl} E_v & ::= & [\,] \mid (V\ E_v) \mid (E_v\ M) \\ E_n & ::= & [\,] \mid (E_n\ M) \end{array}$$

Conceptually, the hole of an evaluation context ($[\,]$) points to the current instruction, which must be a β_v or β redex. The decomposition of M into $E[(V\ N)]$ where $(V\ N)$ is a redex means that the current instruction is $(V\ N)$ and that the rest of the computation, the continuation, is E [16]. For a call-by-value language, the syntax of the terms can therefore be reformulated as follows:

$$\begin{array}{rcll} M & ::= & V \mid E_v[(V\ V)] & (\Lambda) \\ V & ::= & x \mid (\lambda x.M) & (\mathit{Values}) \\ E_v & ::= & [\,] \mid (V\ E_v) \mid (E_v\ M) & (\mathit{EvCont}) \end{array}$$

A similar definition exists for the call-by-name language except that arguments are not evaluated before a function call, *i.e.*, evaluation contexts do not include contexts of the shape $(V\ E_n)$.

Using evaluation contexts, the definitions of the standard reduction functions for call-by-value and call-by-name respectively are as follows:

$$\begin{aligned} E_v[((\lambda x.M)\ V)] &\longmapsto_v E_v[M[x := V]] \\ E_n[((\lambda x.M)\ N)] &\longmapsto_n E_n[M[x := N]] \end{aligned}$$

A complete evaluation applies the single-step functions repeatedly and either reaches an answer or diverges. The notation $\longmapsto^*$ denotes the reflexive, transitive closure of the relation $\longmapsto$. The semantics of Λ is defined as follows:

$$\begin{aligned} eval_v(M) &= V \textit{ if and only if } M \longmapsto_v^* V && \text{(call-by-value)} \\ eval_n(M) &= V \textit{ if and only if } M \longmapsto_n^* V && \text{(call-by-name)} \end{aligned}$$

3. The Origins and Practice of CPS

The idea of transforming programs to "continuation-passing style" first appeared in the mid-sixties.[3] For a few years, the transformation remained part of the folklore of computer science until Fischer and Reynolds codified it in 1972.

The first subsection briefly reviews the original encoding of the CPS transformation. In the second subsection, we analyze the universe of CPS terms and the so-called administrative CPS reductions.

3.1. The Original Encoding

Fischer [21] studied two implementation strategies for Λ: a heap-based retention strategy in which all variable bindings are retained until no longer needed, and a stack-based deletion strategy in which variable bindings are destroyed when control leaves the procedure (or block) in which they were created. He concluded that

> no real power is lost in restricting oneself to a deletion strategy implementation, for any program can be translated into an equivalent one which will work correctly under such an implementation [21, page 104].

[3] The origin of the concept of "continuation" can be traced back to A. van Wijngaarden [45] who defined a source to source transformation that eliminates jumps (goto instructions) from a program in favor of procedures that never return.

The translation he refers to is the Fischer CPS transformation.

Definition 2 (Fischer CPS) *Let k, m, $n \in$ Vars be variables that do not occur in the argument to $\mathcal{F}$.*

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

Reynolds [41] investigated definitional interpreters for higher-order languages. One of his goals was the desire to liberate the definition of a language from the parameter-passing technique of the defining language. He developed a method to transform an interpreter such that it becomes indifferent to whether the underlying parameter passing technique is call-by-value or call-by-name. His transformation is essentially the same transformation as Fischer's.[4] Plotkin [39] later proved Reynolds's ideas correct.

Theorem 1 (Plotkin [39]) *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.

3.2. The Universe of CPS Terms

Since we are interested in the analysis of $\beta\eta$-equality on CPS terms, our universe of discourse consists of all terms that contribute to the proofs of equations like:

$$\boldsymbol{\lambda}\beta\eta \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!].$$

[4] In Reynolds's transformation, the continuation is the second argument to a procedure.

Because the notion of reduction $\beta\eta$ is *CR* [4], it suffices to consider equations of the form:

$$\boldsymbol{\lambda}\beta\eta \vdash \mathcal{F}[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow P.$$

Hence, the universe of discourse for CPS terms is the set:

$$\{P \mid \exists M \in \Lambda.\ \boldsymbol{\lambda}\beta\eta \vdash \mathcal{F}[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow P\}.$$

Unfortunately, this set includes a large number of terms that have no counterpart in the source language. For example, the source reduction:

$$((\lambda x.x)\ y) \longrightarrow y$$

corresponds to the following derivation on CPS terms:

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

The derivation mostly consists of reductions that do not correspond to reductions of source terms.

The "new" reductions on CPS terms are known as *administrative* reductions [39]. In order to give a precise definition of these new reductions, we modify the Fischer CPS transformation by overlining all λ-abstractions that are introduced during the translation. The reduction of any of these overlined λ-abstractions constitutes an administrative reduction.

Definition 3 ($\mathcal{F}$, $\overline{\beta}$, $\overline{\eta}$) *Let $k, m, n \in$ Vars be variables that do not occur in the argument to $\mathcal{F}$.*

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

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

A β- or η-reduction is an administrative reduction if it involves overlined abstractions:

$$((\overline{\lambda} x.M)\ N) \longrightarrow M[x := N] \qquad (\overline{\beta})$$

$$(\overline{\lambda} x.Mx) \longrightarrow M \qquad x \notin FV(M) \qquad (\overline{\eta})$$

To simplify the derivation of the source reductions that correspond to $\beta\eta$ on CPS terms, we split the problem in two parts: finding source reductions that correspond to administrative CPS reductions (Section 4), and finding source reductions that correspond to proper CPS reductions (Sections 5 and 6).

4. A Compacting CPS Transformation

To identify all the source reductions that correspond to the CPS administrative reductions at once, we define a compacting CPS transformation that performs the administrative reductions in the output of $\mathcal{F}$ and produces terms in $\overline{\beta}\overline{\eta}$-normal form.

The first subsection formally defines the process of eliminating the administrative reductions from the output of $\mathcal{F}$ and analyzes the connection between the elimination process and the evaluation of CPS programs. The second subsection includes a new compacting CPS transformation that illuminates the effect of administrative reductions. Subsection 3 includes the source reductions that correspond to the elimination of the administrative CPS reductions. Finally, the last subsection includes the definition of a simplified universe of CPS terms based on the new CPS transformation.

4.1. The Two-Pass CPS Transformation

The relation $\mathcal{F}2$ combines the Fischer CPS transformation with the elimination of the administrative reductions.

Definition 4 ($\mathcal{F}2$) *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.*

Each source term is related by $\mathcal{F}2$ to exactly one CPS term in $\overline{\beta}\overline{\eta}$-normal form.

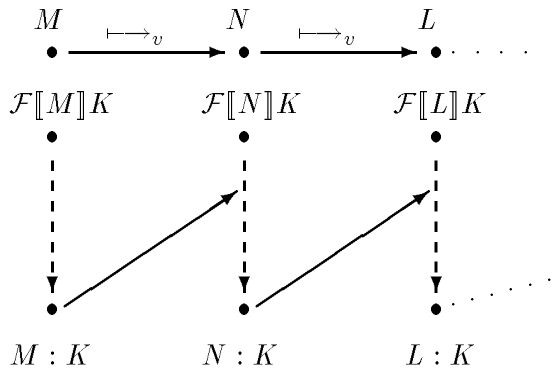
Proposition 2 *The relation $\mathcal{F}2$ is a total function from Λ to Λ.*

Proof: [5] It is sufficient to show that $\mathcal{F}[\![M]\!]$ has a unique $\overline{\beta}\overline{\eta}$-normal form. The $\overline{\beta}$-reductions starting with $\mathcal{F}[\![M]\!]$ are a special kind of reductions called developments [4, section 11.2]. It follows that $\mathcal{F}[\![M]\!]$ has a unique $\overline{\beta}$-normal form [4, corollary 11.2.24]. Moreover, a Λ-term has a β-normal form if and only it has a $\beta\eta$-normal form [4, corollary 15.1.5]. It follows that $\mathcal{F}[\![M]\!]$ has a unique $\overline{\beta}\overline{\eta}$-normal form. ∎

In other words, the function $\mathcal{F}2$ specifies a CPS transformation that produces terms without any administrative redexes.[6]

The proposition also establishes that administrative reductions can be performed in any order without affecting the result. A quick look at their effect on standard reduction sequences is particularly illuminating.

According to Plotkin [39], the standard reduction sequence of a source program relates to the standard reduction sequence of its CPS counterpart as described in the following diagram:



[5]Improved by Robert Harper.

[6]The function $\mathcal{F}2$ eliminates more administrative than other CPS transformations [3, 13, 30, 43, 46]. For example, applying $\mathcal{F}2$ to $(((\lambda x.\lambda y.x)\ a)\ b)$ yields $\lambda k.((\lambda x.((\lambda y.kx)\ b))\ a)$. For the same example, both Steele's Rabbit transformation [46] and the Danvy/Filinski transformation [13] yield the term:

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

Even though this term only contains source redexes, we could still optimize it without eliminating source redexes:

$$\begin{aligned}
\lambda\beta\eta \vdash \lambda k.((\lambda x k_1.(k_1 \lambda y k_2.k_2 x))\ a\ (\lambda m.mbk)) &= \lambda k.((\lambda m.mbk)\ (\lambda y k_2.k_2 a)) \\
&= \lambda k.((\lambda y k_2.k_2 a)\ b\ k) \\
&= \lambda k.ka \\
&= \lambda k.((\lambda y.ka)\ b) \\
&= \lambda k.((\lambda x.((\lambda y.kx)\ b))\ a).
\end{aligned}$$

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.

By Plotkin's analysis, 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.

Although the action of "lifting" the redex $((\lambda x.\cdots)\ z)$ to the top level happens naturally in the CPS framework, it does not require an explicit CPS conversion, *i.e.*, we can rewrite the original program as follows:

$$((\lambda y.y)\ ((\lambda x.x)\ z)) \longrightarrow ((\lambda x.((\lambda y.y)\ x))\ z).$$

This example suggests that administrative CPS reductions are naturally expressible in the source language. We investigate the exact nature of these source reductions in the next subsection.

4.2. The CPS Transformation $\mathcal{C}_k$

By specifying the function $\mathcal{F}2$ in a new and original way that illuminates the effect of administrative reductions on the reduction of source and CPS terms, we can directly identify two notions of reduction on source terms that perform the same task as administrative reductions.

The key insights that are necessary to derive the new CPS transformation are the following:

- The evaluation context is a syntactic representation of the continuation (cf. Section 2).
- The first redex in a standard reduction sequence always occurs inside the evaluation context (cf. Section 2).
- Administrative reductions "lift" the redex that occurs inside the evaluation context to the top level (cf. Section 4.1).

Based on these insights, we develop the new compacting CPS transformation $\mathcal{C}_k$. The transformation relies on the following definition of the set of evaluation contexts, which is more suitable for the following definition:

$$E \ ::= \ [\,] \ \mid E[(V\ [\,])] \ \mid E[([\,]\ M)]$$

The definition generates the same set as the one in Section 2.

Definition 5 ($\mathcal{C}_k, \Phi, \mathcal{K}_k$) *The CPS transformation uses three mutually recursive functions: $\mathcal{C}_k$ to transform terms, Φ to transform values, and $\mathcal{K}_k$ to transform evaluation contexts. Let $k, u_i \in$ Vars be variables that do not occur in the argument to $\mathcal{C}_k$.*[7]

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

The transformation of a complete program M is $\lambda k.\mathcal{C}_k[\![M]\!]$. The function $\mathcal{C}_k$ is parametrized 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

[7]The CPS transformation $\mathcal{C}_k$ is, in spirit, similar to the CPS transformation by Friedman, Wand, and Haynes [24, chapter 8], but differs significantly in its formal part.

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.

While $\mathcal{C}_k$, Φ 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 6 (Size) *The* size *of a term M, $|M|$, is the number of variables in M (including binding occurrences). The* size *of a context E, $|E|$, 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 result also establishes that $\mathcal{C}_k$ is a total function.

Proposition 3 *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:

- $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![M]\!]\; k) = \mathcal{C}_k[\![M]\!]$.
- $\mathcal{C}_k[\![M]\!]$ 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. The proofs of the first two claims are in the Appendix (Page 49). ∎

4.3. Administrative Source Reductions: The A-Reductions

The function $\mathcal{C}_k$ incorporates the reduction of all administrative redexes from the output of the Fischer CPS. Hence, if $\mathcal{F}[\![M]\!]$ and $\mathcal{F}[\![N]\!]$ reduce to a common term by administrative reductions only, $\mathcal{C}_k[\![M]\!]$ is identical to $\mathcal{C}_k[\![N]\!]$. The definition of the function $\mathcal{C}_k$ shows that, in two cases, different inputs are indeed mapped to the same output. (Proposition 11 verifies that there are indeed only two such cases.)

Lemma 4 (β_{lift}, β_{flat}) *Let $M, N, L \in \Lambda$, $E \in \mathit{EvCont}$, and $z \in \mathit{Vars}$:*

$$\begin{array}{rcll} \mathcal{C}_k[\![E[((\lambda x.M)\; N)]]\!] & \equiv & \mathcal{C}_k[\![((\lambda x.E[M])\; N)]\!] & \text{where } x \notin FV(E) \\ \mathcal{C}_k[\![((z\; M)\; L)]\!] & \equiv & \mathcal{C}_k[\![((\lambda u.uL)\; (z\; M))]\!] & \text{where } u \notin FV(L) \end{array}$$

Proof: The proof of the first claim is straightforward. The proof of the second claim[8] is in the Appendix (Page 51). The identity in the second statement holds modulo the decorating overlines above administrative λ-abstractions. ∎

To characterize these effects of the $\mathcal{C}_k$-translation, we introduce two reductions on Λ that capture the effect of the administrative reductions.

Definition 7 **(A-reductions, β_{lift}, β_{flat})** *The set of axioms A contains two reductions:*

$$\begin{array}{rcll} E[((\lambda x.M)\ N)] & \longrightarrow & ((\lambda x.E[M])\ N) & (\beta_{\mathit{lift}}) \\ & & \textit{where } E \neq [\ ] \textit{ and } x \notin FV(E) & \\ ((z\ M)\ L) & \longrightarrow & ((\lambda u.(u\ L))\ (z\ M)) & (\beta_{\mathit{flat}}) \\ & & \textit{where } u \notin FV(L) & \end{array}$$

As Lemma 4 shows, the A-reductions define equivalence classes of source terms that map to the same CPS term.

At this point, the decorating overlines above the special λ-abstractions become irrelevant. In the remainder of the paper, we ignore the distinction between λ and $\overline{\lambda}$.

4.4. The CPS Language

Using the equivalence of the functions $\mathcal{F}2$ and $\mathcal{C}_k$, we can specify the universe of CPS terms as follows:

$$S \stackrel{df}{=} \{P \mid \exists M \in \Lambda.\ \boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow P\},$$

ignoring the outermost binding of the continuation.

The context-free grammar that generates the set S can be directly derived from the right hand sides of the equations in Definition 5. According to the definition, all terms in the CPS language are an application of a continuation to a value. Values are either variables or abstractions that transform continuations. Continuations are either variables, or the result of the application of a value to a continuation, or an abstraction that transforms a value to an answer.

[8] The earlier version of the paper [42] erroneously included an arbitrary term M in place of the variable z.

Definition 8 (CPS grammar, CPS program, $cps(\Lambda)$) *Let $K\text{-}Vars = \{k\}$ be a set of continuation variables such that $Vars \cap K\text{-}Vars = \emptyset$.*

$$\begin{array}{rcll} P & ::= & (K\ W) & (cps(\Lambda) = Answers) \\ W & ::= & x \mid (\lambda k.K) & (cps(Values) = CPS\text{-}Values) \\ K & ::= & k \mid (W\ K) \mid (\lambda x.P) & (cps(EvCont) = Continuations) \end{array}$$

The special status reserved for the variable k ensures that the continuation parameter occurs exactly once in the body of each abstraction $\lambda k.K$. A *program* in CPS form is a closed term of the form $((\lambda k.P)\ (\lambda x.x))$ where k is the special continuation parameter. When working with the quotient of the language under α-equivalence, the special status of the name "k" disappears but the linearity constraint remains.

The following theorem establishes that the two definitions of CPS terms define the same language.

Theorem 5 *$S = cps(\Lambda)$.*

> **Proof:** For the left to right inclusion, it suffices to show that the output of $\mathcal{C}_k$ is a subset of $cps(\Lambda)$, and that the latter language is closed under $\beta\eta$-reductions. We omit the proof of the first claim. The second claim follows from the Subject Reduction Lemma (Lemma 6).
>
> For the opposite implication, *i.e.*, $cps(\Lambda) \subseteq S$, it suffices to show that for all $P \in cps(\Lambda)$, there exists $M \in \Lambda$ such that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow P$. The proof of this auxiliary claim is in the Appendix (Page 52). ∎

To complete the proof of the theorem, we need to establish that $\beta\eta$-reductions on $cps(\Lambda)$ preserve the syntactic categories of the terms.

Lemma 6 (Subject Reduction) *Let $P_1 \in cps(\Lambda)$, $W_1 \in cps(Values)$ and $K_1 \in cps(EvCont)$, then,*

1. *$\boldsymbol{\lambda}\beta\eta \vdash P_1 \longrightarrow P_2$ implies $P_2 \in cps(\Lambda)$.*
2. *$\boldsymbol{\lambda}\beta\eta \vdash W_1 \longrightarrow W_2$ implies $W_2 \in cps(Values)$.*
3. *$\boldsymbol{\lambda}\beta\eta \vdash K_1 \longrightarrow K_2$ implies $K_2 \in cps(EvCont)$.*

> **Proof:** See Appendix (Page 53). ∎

The above lemma implies that $\beta\eta$-reductions on CPS terms can be naturally characterized as reductions that apply to continuations and reductions that apply to values.

Corollary 7 *The reductions β and η on $cps(\Lambda)$ can be decomposed into reductions that apply to values (β_w and η_w) and reductions that apply to continuations (β_k and η_k):*

$$\begin{array}{rcllr}
((\lambda x.P)\ W) & \longrightarrow & P[x := W] & & (\beta_w) \\
((\lambda k.K_1)\ K_2) & \longrightarrow & K_1[k := K_2] & & (\beta_k) \\
(\lambda k.Wk) & \longrightarrow & W & & (\eta_w) \\
(\lambda x.Kx) & \longrightarrow & K & x \notin FV(K) & (\eta_k)
\end{array}$$

5. A CPS Inverse

In order to map the CPS reductions β_w, β_k, η_w, and η_k to reductions on the source language, we define a mapping from $cps(\Lambda)$ to Λ. After presenting the new transformation in the first subsection, we study its connection to the CPS transformation in the second subsection.

5.1. The Transformation $\mathcal{C}^{-1}$

Based on the inductive definition of the CPS language, the specification of an "inverse" to the CPS transformation is almost straightforward: the source term corresponding to $(K\ W)$ is $E[V]$ where E is the evaluation context that syntactically represents the continuation K, and V is the value that corresponds to W.

Definition 9 ($\mathcal{C}^{-1}, \Phi^{-1}, \mathcal{K}^{-1}$) *Let $P \in cps(\Lambda)$, $W \in cps(Values)$, and $K, K_1, K_2 \in cps(EvCont)$:*

$$\begin{array}{rcl}
\mathcal{C}^{-1} : cps(\Lambda) & \rightarrow & \Lambda \\
\mathcal{C}^{-1}[\![(K\ W)]\!] & = & \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]]
\end{array}$$

$$\begin{array}{rcl}
\Phi^{-1} : cps(Values) & \rightarrow & Values \\
\Phi^{-1}[\![x]\!] & = & x \\
\Phi^{-1}[\![(\lambda k.k)]\!] & = & \lambda x.x \\
\Phi^{-1}[\![(\lambda k.WK)]\!] & = & \lambda x.\mathcal{C}^{-1}[\![(W\ K)\ x]\!] \\
\Phi^{-1}[\![(\lambda k.\lambda x.P)]\!] & = & \lambda x.\mathcal{C}^{-1}[\![P]\!]
\end{array}$$

$$\begin{array}{rcl}
\mathcal{K}^{-1} : cps(EvCont) & \rightarrow & EvCont \\
\mathcal{K}^{-1}[\![k]\!] & = & [\,] \\
\mathcal{K}^{-1}[\![(x\ K)]\!] & = & \mathcal{K}^{-1}[\![K]\!][(x\ [\,])] \\
\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!] & = & \mathcal{K}^{-1}[\![K_1[k := K_2]]\!] \\
\mathcal{K}^{-1}[\![(\lambda x.P)]\!] & = & ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\,])
\end{array}$$

The correctness of the function $\mathcal{C}^{-1}$ is subject of the following theorem. The first part of the theorem establishes that the composition of $\mathcal{C}^{-1}$ and $\mathcal{C}_k$ respects $\beta\eta$-equality. The second part of the theorem establishes the stronger property that, when restricted to images of Λ terms, the composition of $\mathcal{C}^{-1}$ and $\mathcal{C}_k$ yields the identity function.

Theorem 8 *Let $P \in cps(\Lambda)$, $K \in cps(EvCont)$. Then,*

1. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] = P$ *and* $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!] = K$*;*
2. $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] \equiv P$ *and* $(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!] \equiv K$ *if there exists $M \in \Lambda$ and $E \in EvCont$ such that $P = \mathcal{C}_k[\![M]\!]$ and $K = \mathcal{K}_k[\![E]\!]$.*

Proof: See Appendix (Page 54). ∎

5.2. Composing $\mathcal{C}_k$ and $\mathcal{C}^{-1}$

The compacting CPS transformation $\mathcal{C}_k$ maps all the members of A-equivalence classes to the same CPS term (cf. Section 4.3). Our inverse CPS transformation maps this CPS term back to a particular element of the equivalence class, the element in $\beta_{lift}\beta_{flat}$-normal form. In order to establish this result, we first define a subset of Λ in $\beta_{lift}\beta_{flat}$-normal form.

Definition 10 (Λ_a) *The language Λ_a is a subset of Λ that only includes terms in $\beta_{lift}\beta_{flat}$-normal form.*

$$\begin{array}{rcll}
M & ::= & E[V] & (\Lambda_a) \\
V & ::= & x \mid (\lambda x.M) & (Values_a) \\
E & ::= & [\,] \mid ((\lambda x.M)\ [\,]) \mid E[(x\ [\,])] & (EvCont_a)
\end{array}$$

We omit the simple inductive proof that the elements of the language are actually in $\beta_{lift}\beta_{flat}$-normal form.

The range of the function $\mathcal{C}^{-1}$ is included in Λ_a, *i.e.*, any output of $\mathcal{C}^{-1}$ is in $\beta_{lift}\beta_{flat}$-normal form.

Lemma 9 *Let* $P \in cps(\Lambda)$ *and* $K \in cps(EvCont)$, *then* $\mathcal{C}^{-1}[\![P]\!] \in \Lambda_a$ *and* $\mathcal{K}^{-1}[\![K]\!] \in EvCont_a$.

> **Proof:** See Appendix (Page 56). ∎

With the help of this lemma, we can now specify the precise relation between the CPS transformation and its inverse. The effect of composing the CPS transformation with its inverse is to reduce terms to $\beta_{lift}\beta_{flat}$-normal form. Naturally, if a term is already in $\beta_{lift}\beta_{flat}$-normal form, then the composition yields the identity function.

Theorem 10 *Let* $M \in \Lambda$, *then:*

1. $\boldsymbol{\lambda}\beta_{lift}\beta_{flat} \vdash M \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$,
2. $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv M$ *if there exists* $P \in cps(\Lambda)$ *such that* $M = \mathcal{C}^{-1}[\![P]\!]$.

> **Proof:** See Appendix (Page 56). ∎

Put differently, the theorem asserts that the reductions $\beta_{lift}\beta_{flat}$ capture all possible equivalences introduced by administrative reductions. If the CPS transforms of M and N are related by those administrative reductions that $\mathcal{C}_k$ eliminates, then it must be the case that M and N are related by the axioms $\beta_{lift}\beta_{flat}$.

Proposition 11 *If* $\mathcal{C}_k[\![M]\!] \equiv \mathcal{C}_k[\![N]\!]$, *then* $\boldsymbol{\lambda}\beta_{lift}\beta_{flat} \vdash M = N$.

> **Proof:** Assume $\mathcal{C}_k[\![M]\!] \equiv \mathcal{C}_k[\![N]\!] \equiv P$. The function $\mathcal{C}^{-1}$ maps P, the CPS transform of M or N, to a source term L. By Theorem 10, both M and N reduce to L by $\beta_{lift}\beta_{flat}$-reductions. It follows that $\boldsymbol{\lambda}\beta_{lift}\beta_{flat} \vdash M = N$. ∎

6. Equational Correspondence for the Pure Language

Using the partial inverse of the CPS transformation, we can systematically derive a set of additional axioms B for $\boldsymbol{\lambda}\beta_v$ such that $X = A \cup B$, *i.e.*, such that $\boldsymbol{\lambda}\beta_v AB$ is complete for $\beta\eta$ reasoning about CPS programs. Once we have the new axiom set, we prove its soundness in the second subsection. In the last subsection, we briefly discuss the correspondence of the calculi.

6.1. Completeness

As specified in Corollary 7, the possible β- and η-reductions on CPS terms are β_w, β_k, η_w, and η_k. To illustrate our technique, we first outline the derivation of reductions corresponding to η_k. Let $(\lambda x.K x) \longrightarrow K$ where $x \notin FV(K)$. Applying $\mathcal{K}^{-1}$ to both sides of the reduction, we get:

$$((\lambda x.\mathcal{C}^{-1}[\![K x]\!])\ [\,]) \qquad \text{and} \qquad \mathcal{K}^{-1}[\![K]\!].$$

To understand how the left hand side could reduce to the right hand side, we proceed by case analysis on K:

- $K = k$: the reduction becomes $((\lambda x.x)\ [\,]) \longrightarrow [\,]$. Since the empty context generally stands for an arbitrary term, the extended set of axioms should therefore contain the reduction:
$$((\lambda x.x)\ M) \longrightarrow M \qquad (\beta_{id})$$
- $K = (y\ K_1)$: the reduction becomes $((\lambda x.\mathcal{K}^{-1}[\![K_1]\!][(y\ x)])\ [\,]) \longrightarrow \mathcal{K}^{-1}[\![K_1]\!][(y\ [\,])]$. By a similar argument as in the first case, we must add the following reduction to the set B:
$$((\lambda x.E[(y\ x)])\ M) \longrightarrow E[(y\ M)] \qquad (\beta_\Omega)$$
- $K = ((\lambda k.K_1)\ K_2)$ or $K = \lambda y.P$: these cases do not introduce any new reductions.

The cases for the other reductions on CPS terms are similar. The resulting set of source reductions X includes all the previously derived reductions and η_v: see Figure 2. The equational theory generated by the full set of

$((\lambda x.M)\ V) \longrightarrow M[x := V]$			(β_v)
$(\lambda x.V x) \longrightarrow V$	$x \notin FV(V)$		(η_v)
$E[((\lambda x.M)\ N)] \longrightarrow ((\lambda x.E[M])\ N)$	$x \notin FV(E), E \neq [\,]$		(β_{lift})
$((z\ M)\ L) \longrightarrow ((\lambda u.(u\ L))\ (z\ M))$	$u \notin FV(L)$		(β_{flat})
$((\lambda x.x)\ M) \longrightarrow M$			(β_{id})
$((\lambda x.E[(y\ x)])\ M) \longrightarrow E[(y\ M)]$	$x \notin FV(E[y])$		(β_Ω)

Figure 2: Source Reductions: $X \stackrel{df}{=} \{\eta_v, \beta_{lift}, \beta_{flat}, \beta_{id}, \beta_\Omega\}$.

axioms $\boldsymbol{\lambda}\beta_v X$ corresponds to Moggi's untyped computational λ-calculus as it appeared in his original Edinburgh LFCS Technical Report [36].

The Completeness Lemma summarizes the connection between the notions of reductions on $cps(\Lambda)$ and the new reductions.

Lemma 12 (Completeness) *Let $P \in cps(\Lambda)$.*

1. *If $\boldsymbol{\lambda}\eta_k \vdash P \longrightarrow Q$ then $\boldsymbol{\lambda}\beta_v\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![Q]\!]$.*
2. *If $\boldsymbol{\lambda}\beta_k \vdash P \longrightarrow Q$ then $\mathcal{C}^{-1}[\![P]\!] \equiv \mathcal{C}^{-1}[\![Q]\!]$.*
3. *If $\boldsymbol{\lambda}\eta_w \vdash P \longrightarrow Q$ then $\boldsymbol{\lambda}\eta_v \vdash \mathcal{C}^{-1}[\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![Q]\!]$.*
4. *If $\boldsymbol{\lambda}\beta_w \vdash P \longrightarrow Q$ then $\boldsymbol{\lambda}\beta_v\beta_{lift}\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![Q]\!]$.*

Proof: The proof of each case is independent from the proofs of other cases.

1. η_k-reduction: The proof is outlined at the beginning of the section.
2. β_k-reduction: By the definition of $\mathcal{K}^{-1}$, $\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!] \equiv \mathcal{K}^{-1}[\![K_1[k := K_2]]\!]$.
3. η_w-reduction: Applying Φ^{-1} to the left hand side, we get the term $\Phi^{-1}[\![(\lambda k.Wk)]\!]$ which is equivalent to $(\lambda x.\mathcal{C}^{-1}[\![((Wk)\ x)]\!])$. We show that the latter term reduces to $\Phi^{-1}[\![W]\!]$ by cases:
   - $W = z$: then the reduction becomes the η_v-reduction: $(\lambda x.zx) \longrightarrow x$.
   - $W = \lambda k.k$: then both sides of the reduction are identical.
   - $W = \lambda k.W_1K$: then again both sides of the reduction are identical.
   - $W = \lambda k.\lambda z.P$: $\lambda x.((\lambda z.\mathcal{C}^{-1}[\![P]\!])\ x) \longrightarrow \lambda z.\mathcal{C}^{-1}[\![P]\!]$ is an η_v-reduction.
4. β_w-reduction: See Appendix (Page 58).

∎

The Completeness Theorem is a direct consequence of the above results.

Theorem 13 (Completeness) *Let $P \in cps(\Lambda)$. If $\boldsymbol{\lambda}\beta\eta \vdash P \longrightarrow\!\!\!\!\!\rightarrow Q$ then $\boldsymbol{\lambda}\beta_vX \vdash \mathcal{C}^{-1}[\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![Q]\!]$.*

Proof: By pasting together the proofs of the Completeness Lemma. ∎

6.2. Soundness

The set of source reductions in Figure 2 is sound with respect to the equational theory over CPS terms. In fact, we can prove the following stronger results on the correspondence of reduction steps.

Lemma 14 (Soundness) *Let* $M \in \Lambda$.

1. *If* $\boldsymbol{\lambda}\beta_v \vdash M \longrightarrow N$ *then* $\boldsymbol{\lambda}\beta \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![N]\!]$.
2. *If* $\boldsymbol{\lambda}\eta_v \vdash M \longrightarrow N$ *then* $\boldsymbol{\lambda}\eta_w\eta_k \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![N]\!]$.
3. *If* $\boldsymbol{\lambda}\beta_{lift} \vdash M \longrightarrow N$ *then* $\mathcal{C}_k[\![M]\!] \equiv \mathcal{C}_k[\![N]\!]$.
4. *If* $\boldsymbol{\lambda}\beta_{flat} \vdash M \longrightarrow N$ *then* $\mathcal{C}_k[\![M]\!] \equiv \mathcal{C}_k[\![N]\!]$.
5. *If* $\boldsymbol{\lambda}\beta_{id} \vdash M \longrightarrow N$ *then* $\boldsymbol{\lambda}\eta_k \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![N]\!]$.
6. *If* $\boldsymbol{\lambda}\beta_\Omega \vdash M \longrightarrow N$ *then* $\boldsymbol{\lambda}\eta_k \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![N]\!]$.

Proof: The proofs for β_{lift} and β_{flat} are in Lemma 4. The proof for β_v-reductions is in the Appendix (Page 61). The other proofs are similar. ∎

The Soundness Theorem summarizes the results of this subsection.

Theorem 15 (Soundness) *If* $\boldsymbol{\lambda}\beta_v X \vdash M \longrightarrow\!\!\!\!\!\rightarrow N$ *then* $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![N]\!]$.

Proof: By pasting the proofs of the Soundness Lemma. ∎

6.3. Equational Correspondence

The Completeness and Soundness Theorems in the previous sections are formulated in the most precise way. In particular, the theorems relate *reduction* steps in one calculus to *reduction* steps in the other calculus. Together with Theorems 8 and 10 about the composition of the CPS transformation and its inverse, they imply the results of Figure 3. In the figure, the dotted lines correspond to the application of $\mathcal{C}_k$ or $\mathcal{C}^{-1}$. The solid lines represent sequences of reductions.

The correspondence of reduction steps reveals the close relation between source terms in A-normal form and CPS terms. Unfortunately, this correspondence of reduction steps relies crucially on the properties of the functions $\mathcal{C}_k$ and $\mathcal{C}^{-1}$, and does not appear to hold for arbitrary CPS transformation and their inverses.

$$
\begin{tikzcd}
M \arrow[rrr, "\beta_v X"'] \arrow[dr, "\beta_{lift}\beta_{\mathit{flat}}"] & & & N \arrow[dl, "\beta_{lift}\beta_{\mathit{flat}}"'] \\
& (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \arrow[r, "\beta_v X"] \arrow[d, dashed, leftrightarrow] & (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![N]\!] \arrow[d, dashed, leftrightarrow] & \\
& \mathcal{C}_k[\![M]\!] \arrow[r, "\beta\eta"] & \mathcal{C}_k[\![N]\!] &
\end{tikzcd}
$$

Figure 3: The Correspondence of Reduction Steps.

In contrast, the correspondence of equalities in the source and CPS calculi holds for any transformations *cps* and *uncps* that satisfy the following equations:

$$
\begin{array}{rcl}
\boldsymbol{\lambda}\beta\eta & \vdash & cps(M) = \mathcal{C}_k[\![M]\!] \\
\boldsymbol{\lambda}\beta_v X & \vdash & uncps(P) = \mathcal{C}^{-1}[\![P]\!]
\end{array}
\qquad (\dagger)
$$

For such transformations, it is straightforward to deduce variants of Theorems 8, 10, 15, and 13 that relate equalities in one calculus to equalities in the other calculus. The combination of the four theorems implies an equational correspondence in the sense of Definition 1.

Theorem 16 (Equational Correspondence) *The theory $\boldsymbol{\lambda}\beta_v X$ equationally corresponds to the theory $\boldsymbol{\lambda}\beta\eta$ for any cps and uncps functions satisfying* ($\dagger$).

The formulation of the calculus $\boldsymbol{\lambda}\beta_v X$ based on the six axioms in Figure 2 is only necessary for the correspondence of reduction steps. For the equational correspondence, there is no reason to distinguish between the reductions β_{flat}, β_{id}, and β_Ω as we can summarize the three reductions with the following axiom:

$$((\lambda x.E[x])\ M) \ = \ E[M] \qquad x \notin FV(E) \qquad (\beta'_\Omega)$$

We will use the axiom (β'_Ω) in the remainder of the paper.

7. Non-Local Control Operators

After establishing the Equational Correspondence Theorem for pure call-by-value languages, we turn our attention to languages with control operators. Specifically, we investigate the addition of the control operators *abort* ($\mathcal{A}$) and *call-with-current-continuation* (*callcc*). Informally, $\mathcal{A}$ permits the programmer to ignore the rest of a computation and return the value of a subexpression as the result of the entire program, while *callcc* provides the programmer with a procedural abstraction of the rest of the computation. Because the two operators manipulate the global control state of the program, their CPS transforms are procedures that manipulate the continuation in non-standard ways. As a consequence, the CPS language includes new terms and the first Equational Correspondence Theorem no longer applies. In the remainder of the section, we formalize these ideas and conclude with a version of the Equational Correspondence Theorem for the extended language. The development of the section follows the development of Sections 4 to 6 with one exception. None of the intermediate results is concerned with mapping the *reductions* of one calculus to the *reductions* of the other. Rather the intermediate results only relate the equalities of one calculus to the equalities of the other. At this point, it is an open question whether the results can be re-established for reductions (as opposed to equalities).[9]

7.1. The Extended Language and its Semantics

The extension of the source language with the functional constants *callcc* and $\mathcal{A}$ results in the language $\Lambda + callcc + \mathcal{A}$:

$$\begin{aligned} M &::= V \mid E[(V\ V)] \\ V &::= x \mid (\lambda x.M) \mid callcc \mid \mathcal{A} \\ E &::= [\,] \mid (V\ E) \mid (E\ M) \end{aligned}$$

Instead of providing a formal semantics for *callcc* and $\mathcal{A}$ in terms of standard reductions, we follow the more traditional route and immediately specify

[9] Griffin [26] established that, in the presence of control operators similar to $\mathcal{A}$ and *callcc*, standard reductions on source terms correspond to standard reductions on CPS terms. His result does not imply that an arbitrary sequence of reductions in either calculus correspond to another sequence of reductions in the other calculus.

the translation of these values into CPS form and use this translation as their formal semantics.[10] The extensions to $\mathcal{C}_k$ (or $\mathcal{F}$) consist of two additional clauses to the function Φ (or $\mathcal{F}_v$) [8]:

$$\begin{aligned}\Phi[\![callcc]\!] &= (\lambda k.\lambda u.((u\ k)\ \lambda d.k)) \\ \Phi[\![\mathcal{A}]\!] &= (\lambda k.\lambda x.x)\end{aligned}$$

The CPS transform of *callcc* is a procedure that expects a continuation k and an argument u. The non-standard manipulation of the continuation is manifest in the second argument to u, which is a procedural abstraction of the continuation. Similarly, the CPS transform of $\mathcal{A}$ is a procedure that expects a continuation k and an argument x. The procedure ignores its continuation argument (k) and immediately returns its value argument (x). The non-use of k is again a non-standard manipulation of the continuation. Given the CPS transformation, the formal semantics of the source language is:

$$eval_v(M) = V \quad \textit{if and only if} \quad eval_n((\lambda k.\mathcal{C}_k[\![M]\!])\ \lambda x.x) = \Phi[\![V]\!].$$

In order to simplify the following discussions (and proofs), we use a CPS transformation that is less compacting but more suited for the analysis than the one in Definition 5.

Definition 11 ($\mathcal{C}_k$ with Control Operators) *Let $k, u_i \in$ Vars be variables that do not occur in the argument to $\mathcal{C}_k$.*

$$\begin{aligned}\mathcal{C}_k[\![V]\!] &= (k\ \Phi[\![V]\!]) \\ \mathcal{C}_k[\![E[(V_1\ V_2)]]\!] &= ((\Phi[\![V_1]\!]\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V_2]\!]) \\ \\ \Phi[\![x]\!] &= x \\ \Phi[\![\lambda x.M]\!] &= \lambda k.\lambda x.\mathcal{C}_k[\![M]\!] \\ \Phi[\![callcc]\!] &= \lambda k.\lambda u.((u\ k)\ \lambda d.k) \\ \Phi[\![\mathcal{A}]\!] &= \lambda k.\lambda x.x \\ \\ \mathcal{K}_k[\![[\ ]]\!] &= k \\ \mathcal{K}_k[\![E[(V\ [\ ])]]\!] &= (\Phi[\![V]\!]\ \mathcal{K}_k[\![E]\!]) \\ \mathcal{K}_k[\![E[([\ ]\ M)]]\!] &= \lambda f.\mathcal{C}_k[\![E[(f\ M)]]\!]\end{aligned}$$

[10] Felleisen *et al.* [16, 18, 19] and Talcott [48] use alternative definitions that do not rely on the CPS transformation.

Besides the extensions to *callcc* and $\mathcal{A}$, the transformation differs from the function in Definition 5 in the following aspect. The new function translates expressions of the form $E[(V\ M)]$ uniformly for all values V. By not including a special clause for each kind of value, the CPS transformation may produce terms with administrative redexes. However, the presence of these administrative redexes is irrelevant since it does not affect the set of reachable CPS terms, and we are no longer concerned with mapping the reductions of one calculus to the reductions of the other calculus.

7.2. The CPS Language and the Inverse Translation

The closure of the output of $\mathcal{C}_k$ under $\beta\eta$-reductions yields an extension of the CPS language of Definition 8.

Definition 12 (CPS grammar $cps(\Lambda + callcc + \mathcal{A})$**)** *The extended CPS terms are generated by the following grammar:*

$$\begin{array}{rcll} P & ::= & W \mid (K\ W) & (cps(\Lambda + callcc + \mathcal{A}) = \mathit{Answers}) \\ W & ::= & x \mid (\lambda k.K) & (\mathit{Values}) \\ K & ::= & k \mid (\lambda x.P) \mid (W\ K) & (\mathit{Continuations}) \end{array}$$

$$\begin{array}{rcl} x & \in & \mathit{Vars} \\ k & \in & K\text{-}\mathit{Vars} = \{k_1, k_2, \ldots\} \textit{ and } K\text{-}\mathit{Vars} \cap \mathit{Vars} = \emptyset \end{array}$$

A comparison with Definition 8 explains how the addition of *callcc* and $\mathcal{A}$ affects the set of reachable CPS terms and thus, how it affects the semantics. Intuitively, *callcc* permits the programmer to "label" arbitrary points in the program. Thus more than one continuation can potentially be lexically visible at any point during the execution of the program. The extended CPS language accommodates this fact by providing an infinite set of continuation variables instead of a singleton. The addition of $\mathcal{A}$ permits the programmer to ignore the current continuation by returning a value as the answer of the entire program. This extension is reflected in the CPS language by extending the syntactic category of answers to include values directly. An equivalent way to understand the effect of $\mathcal{A}$ is that $\mathcal{A}$ ignores the current continuation and uses the initial continuation $(\lambda x.x)$ instead. We therefore may extend the syntactic category of continuations with an "initial continuation" $(\lambda x.x)$. Because, our CPS language is closed under $\beta\eta$-reductions, the addition of the initial continuation results in programs of the shape $P = ((\lambda x.x)\ W) \longrightarrow W$ and thus extends the syntactic category of answers with values.

The inverse CPS transformation mapping the extended CPS language to $\Lambda + callcc + \mathcal{A}$ is the following.

Definition 13 *Let $P \in cps(\Lambda + callcc + \mathcal{A})$. Let W and K be values and continuations in the same language:*

$$\begin{array}{rcl}
\mathcal{C}^{-1}[\![W]\!] & = & (\mathcal{A}\ \Phi^{-1}[\![W]\!]) \\
\mathcal{C}^{-1}[\![(K\ W)]\!] & = & \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]] \\
\\
\Phi^{-1}[\![x]\!] & = & x \\
\Phi^{-1}[\![\lambda k.K]\!] & = & \lambda z.callcc\ \lambda k.\mathcal{K}^{-1}[\![K]\!][z] \\
\\
\mathcal{K}^{-1}[\![k]\!] & = & (k\ [\ ]) \\
\mathcal{K}^{-1}[\![(W\ K)]\!] & = & \mathcal{K}^{-1}[\![K]\!][(\Phi^{-1}[\![W]\!]\ [\ ])] \\
\mathcal{K}^{-1}[\![\lambda x.P]\!] & = & ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\ ])
\end{array}$$

The transformation differs from the function in Definition 9 in several aspects. First, the inverse of an answer W is a term that aborts with the value $\Phi^{-1}[\![W]\!]$. Second, a binding of a continuation k in the CPS language corresponds to a capture of the continuation k in the source language. Finally, every continuation is explicitly invoked. The last two changes exploit an idea due to Danvy and Lawall [14].[11]

The discovery of the call-by-value axioms that correspond to $\beta\eta$-reductions on CPS terms proceeds in the same manner as for the pure language. The resulting axioms consist of the axioms X for the pure language and the control specific axioms in Figure 4. The new axioms have the following intuitive explanation. The first axiom shows that the current continuation is always implicitly applied. The axiom C_{elim} is a garbage-collection rule: continuations captured but not used can be collected. The axiom C_{lift} characterizes the capture of continuations via *callcc* while the axiom C_{abort} shows that continuations abort their context upon invocation. The last *callcc* axiom implies that the continuation of an application is indistinguishable from the continuation of the body.[12] The operator $\mathcal{A}$ eliminates evaluation contexts.

The set of axioms $C_{current}, C_{elim}, C_{lift}, C_{abort}$ and *Abort* constitute the control-specific axioms of the λ_v-C-calculus [18, 19]. The full theory $\boldsymbol{\lambda}\beta_v XC$

[11] Danvy and Lawall [14] perform a counting analysis to determine whether a continuation is used in a non-standard way and include a *callcc* only when necessary. This analysis is unnecessary for our purposes. The outputs of our inverse transformation are provably equal to their outputs (in our axiom system), thus achieving the same effect without the counting analysis.

[12] ... and a good implementation would use the same continuation: we can also interpret this law as imposing tail-call optimization on faithful implementations.

$$
\begin{array}{rcll}
(\mathit{callcc}\ \lambda k.kM) & = & (\mathit{callcc}\ \lambda k.M) & (C_{current}) \\
(\mathit{callcc}\ \lambda d.M) & = & M \qquad d \notin FV(M) & (C_{elim}) \\
E[(\mathit{callcc}\ M)] & = & \mathit{callcc}\ \lambda k.E[(M\ (\lambda f.(k\ E[f])))] & (C_{lift}) \\
 & & k, f \notin FV(E, M) & \\
\mathit{callcc}\ \lambda k.C[E[(k\ M)]] & = & \mathit{callcc}\ \lambda k.C[(k\ M)] \qquad k \notin \mathit{trap}(C) & (C_{abort}) \\
(\mathit{callcc}\ \lambda k.((\lambda x.M)\ N)) & = & ((\lambda x.(\mathit{callcc}\ \lambda k.M))\ N) \qquad k \notin FV(N) & (C_{tail}) \\
E[(\mathcal{A}\ M)] & = & (\mathcal{A}\ M) & (\mathit{Abort})
\end{array}
$$

Figure 4: The Axioms C: $\{C_{current}, C_{elim}, C_{lift}, C_{abort}, C_{tail}, \mathit{Abort}\}$.

corresponds to the restriction of the theory IOCC [48] to the language $\Lambda + \mathit{callcc} + \mathcal{A}$.

The relationship of $\mathcal{C}_k$ to the function $\mathcal{C}^{-1}$ is subject of two lemmas. The first lemma establishes that a CPS term P is $\beta\eta$-equal to $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$ if it contains no free continuation variables, *e.g.*, $P \equiv x$. The lemma proves a more general result that accounts for terms with free variables. The generalization is necessary for CPS terms like $P \equiv (k_1\ x)$ or $P \equiv (k_1\ (\lambda d.k_2))$.

Lemma 17 *Let $P \in cps(\Lambda + \mathit{callcc} + \mathcal{A})$ and K be a continuation, and W be a value in the same language. Also, let $k_1, \ldots, k_n$ be the free continuation variables in these terms, and $k \notin \{k_1, \ldots, k_n\}$. Then,*

1. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = P.$
2. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = K.$
3. $\boldsymbol{\lambda}\beta\eta \vdash (\Phi \circ \Phi^{-1})[\![W]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = W.$

Proof: See Appendix (Page 62). ∎

The second lemma relates the terms M and $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$ via the axioms X for the pure language and the control-specific axioms C in Figure 4.

Lemma 18 *Let $M \in \Lambda + \mathit{callcc} + \mathcal{A}$, E be an evaluation context in the same language, and k a variable that is not free in either. Then,*

- $\boldsymbol{\lambda}\beta_v XC \vdash (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] = (k\ M)$
- $\boldsymbol{\lambda}\beta_v XC \vdash (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!] = (k\ E)$

Proof: See Appendix (Page 63). ∎

7.3. Equational Correspondence

We establish that the calculus $\boldsymbol{\lambda}\beta_v XC$ proves all the equations that $\beta\eta$ can prove on CPS terms. The key lemma is the following Completeness Lemma.

Lemma 19 *Let $P \in cps(\Lambda + callcc + \mathcal{A})$, and let $k_1 \ldots k_n$ be the free continuation variables in P. Then, $\boldsymbol{\lambda}\beta\eta \vdash P = Q$ implies that:*

$$\boldsymbol{\lambda}\beta_v XC \vdash callcc\ \lambda k_1.\ldots callcc\ \lambda k_n.\mathcal{C}^{-1}[\![P]\!] = \\ callcc\ \lambda k_1.\ldots callcc\ \lambda k_n.\mathcal{C}^{-1}[\![Q]\!].$$

Proof: We consider each notion of reduction separately.

1. The reduction is: $((\lambda x.P)\ W) \longrightarrow P[x := W]$. The proof of this case is in the Appendix (Page 64).
2. The reduction is: $(\lambda k.Wk) \longrightarrow W$ where k is not free in W. We can apply Φ^{-1} or $\mathcal{C}^{-1}$ to both sides of the equation since W can be an answer or a value.
   - W is an answer. Then,
   $$\begin{array}{rcl} \mathcal{C}^{-1}[\![(\lambda k.Wk)]\!] & = & (\mathcal{A}\ \lambda z.callcc\ \lambda k.\mathcal{K}^{-1}[\![k]\!][(\Phi^{-1}[\![W]\!]\ z)]) \\ & = & (\mathcal{A}\ \lambda z.\Phi^{-1}[\![W]\!]\ z) \\ & = & (\mathcal{A}\ \Phi^{-1}[\![W]\!]) \end{array}$$
   - W is a value. Then
   $$\Phi^{-1}[\![\lambda k.Wk]\!] \equiv \lambda z.callcc\ \lambda k.\mathcal{K}^{-1}[\![k]\!][(\Phi^{-1}[\![W]\!]\ z)]$$
   and a similar argument as in the preceding subcase applies.
3. The reduction is: $(\lambda x.Kx) \longrightarrow K$ where x is not free in K. Then $\mathcal{K}^{-1}[\![\lambda x.Kx]\!] \equiv ((\lambda x.\mathcal{K}^{-1}[\![K]\!][x])\ [\ ])$ is equal to $\mathcal{K}^{-1}[\![K]\!]$ by β'_Ω.
4. The reduction is: $((\lambda k.K_1)\ K_2) \longrightarrow K_1[k := K_2]$. The proof is in the Appendix (Page 66).

∎

The soundness of the new axioms is the subject of the following lemma.

Lemma 20 *Let $M, N \in \Lambda + callcc + \mathcal{A}$, then $\boldsymbol{\lambda}\beta_v XC \vdash M = N$ implies that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$.*

The above lemmas imply that the calculi $\boldsymbol{\lambda}\beta_v XC$ and $\boldsymbol{\lambda}\beta\eta$ satisfy an Equational Correspondence Theorem.

Theorem 21 (Equational Correspondence) *The theory $\boldsymbol{\lambda}\beta_v XC$ equationally corresponds to the theory $\boldsymbol{\lambda}\beta\eta$ for any cps and uncps functions satisfying (†).*

8. A Realistic Language

To be useful, a Correspondence Theorem should hold in the presence of programming languages constructs such as conditionals, constants, and mutable data objects. To address this issue, we consider a small, but typical subset of Scheme, Core Scheme, that we define in the first subsection. The second subsection introduces the CPS language for Core Scheme. Unlike in previous sections, the CPS language cannot be a subset of Λ, and as a result, the CPS equational theory includes more axioms than just β and η. Finally, in the last subsection we discuss how the different constructs affect the Correspondence Theorems. The development in the section includes enough details so that it can be adapted to the reader's favorite language.

8.1. Core Scheme: *CS*

Core Scheme's basic constants include numerals and booleans; functional constants include operations to manipulate the basic constants, *e.g.*, addition, as well as operations to create, access, and update data-structures, *e.g.*, lists, reference cells. The formal definition extends the grammar in Section 7.1:

$$
\begin{array}{rcl}
M & ::= & \ldots \mid E[(\mathbf{if}\ V\ M\ M)] \mid E[(O\ V)] \mid E[(O\ V\ V)] \\
V & ::= & \ldots \mid c \\
E & ::= & \ldots \mid (\mathbf{if}\ E\ M\ M) \mid (O\ E) \mid (O\ E\ M) \mid (O\ V\ E) \\
c & ::= & \mathsf{true} \mid \mathsf{false} \mid \ulcorner n \urcorner \\
O & ::= & \mathsf{integer?} \mid \mathsf{add1} \mid / \mid \mathsf{ref} \mid \mathsf{deref} \mid \mathsf{setref!} \\
\\
n & \in & \mathbb{N}
\end{array}
$$

Informally, integer? recognizes integers, add1 denotes the increment function, / is the integer division operator, ref creates a reference cell, deref returns the contents of a reference cell, and setref! updates the contents of its first argument (a reference cell) with the value of its second argument.

The set of axioms F in Figure 5 specifies the semantics of the new constructs. For convenience, we write $\rho\{(x_1, V_1), \ldots, (x_n, V_n)\}.M$ as an abbreviation for:

$$
\begin{array}{l}
(\ldots((\lambda x_1. \cdots \lambda x_n.(\mathbf{begin}\ (\mathsf{setref!}\ x_1\ V_1)\ \ldots\ (\mathsf{setref!}\ x_n\ V_n)\ M)) \\
\quad (\mathsf{ref}\ 0)) \\
\ \ldots \\
(\mathsf{ref}\ 0))
\end{array}
$$

where (**begin** M_1 ... M_n) is itself an abbreviation for:

$$(\ldots((\lambda x_1.\cdots\lambda x_n.x_n)\ M_1)\ \ldots\ M_n)$$

$$\begin{array}{rcll}
(\textbf{if}\ \mathsf{true}\ M\ N) & = & M & (\mathit{If}_t)\\
(\textbf{if}\ \mathsf{false}\ M\ N) & = & N & (\mathit{If}_f)\\
\\
(\mathsf{add1}\ \ulcorner n\urcorner) & = & \ulcorner n+1\urcorner & (\mathit{Add})\\
(/\ \ulcorner n\urcorner\ \ulcorner m\urcorner) & = & \ulcorner n/m\urcorner \qquad m \neq 0 & (\mathit{Div})\\
(/\ \ulcorner n\urcorner\ \ulcorner 0\urcorner) & = & (\mathcal{A}\ \ulcorner 5\urcorner) & (\mathit{Div}_e)\\
\\
(\mathsf{integer?}\ \ulcorner n\urcorner) & = & \mathsf{true} & (\mathit{Int}_t)\\
(\mathsf{integer?}\ V) & = & \mathsf{false} \qquad V \neq \ulcorner n\urcorner & (\mathit{Int}_f)\\
\\
(\mathsf{ref}\ V) & = & \rho\{(x,V)\}.x \qquad x \notin FV(V) & (\mathit{ref})\\
\rho\theta\cup\{(x,V)\}.E[(\mathsf{deref}\ x)] & = & \rho\theta\cup\{(x,V)\}.E[V] & (\mathit{deref})\\
\rho\theta\cup\{(x,V_1)\}.E[(\mathsf{setref!}\ x\ V_2)] & = & \rho\theta\cup\{(x,V_2)\}.E[V_2] & (\mathit{setref})\\
\rho\theta_1.\rho\theta_2.M & = & \rho\theta_1\cup\theta_2.M & (\mathit{ref}_\cup)
\end{array}$$

Figure 5: Additional Axioms F for CS.

The first three sets of axioms are straightforward; the last set specifies the semantics of reference cells [9, 33].

The combination of the equational theory $\boldsymbol{\lambda}\beta_v XC$ with the new axioms F results in an inconsistent equational system due to η_v. For example, using η_v, we can show that for any two terms M and N:

$$\begin{array}{rcll}
M & = & (\mathsf{if\ true}\ M\ N) & (\mathit{If}_t)\\
 & = & (\mathsf{if}\ (\mathsf{integer?}\ \ulcorner 0\urcorner)\ M\ N) & (\mathit{Int}_t)\\
 & = & (\mathsf{if}\ (\mathsf{integer?}\ (\lambda x.(\ulcorner 0\urcorner\ x)))\ M\ N) & (\eta_v)\\
 & = & (\mathsf{if\ false}\ M\ N) & (\mathit{Int}_f)\\
 & = & N & (\mathit{If}_f)
\end{array}$$

To avoid the consistency problem, we restrict the equational theory for the source language by eliminating η_v.

8.2. The CPS Language: CS_k

Given our extensions to the source language, the CPS language cannot be a subset of Λ unless the CPS transformation encodes all new constructs in the source language using procedures. Since such an encoding is not a proper CPS transformation, the CPS language must include constructs that correspond to the extensions of the source language.

The set of CPS terms extends the language in Definition 12 with the following additional clauses:

$$\begin{array}{rcl}
P & ::= & \ldots \mid (\textbf{if}\ W\ P\ P) \mid (O_k\ K\ W) \mid (O_k\ K\ W\ W) \\
W & ::= & \ldots \mid c \\
K & ::= & \ldots \\
c & ::= & \mathsf{true} \mid \mathsf{false} \mid \ulcorner n \urcorner \\
O_k & ::= & \mathsf{integer?}_k \mid \mathsf{add1}_k \mid /_k \mid \mathsf{ref}_k \mid \mathsf{deref}_k \mid \mathsf{setref!}_k
\end{array}$$

The semantics of the new constructs in the CPS language matches the semantics of the corresponding constructs in the source language. The set of axioms F_k in Figure 6 specifies this semantics precisely. For convenience, we also use $\rho\{(x_1, W_1), \ldots, (x_n, W_n)\}.P$ as an abbreviation for:

$$\begin{array}{l}
(\mathsf{ref}_k\ (\lambda x_1. \cdots\ (\mathsf{ref}_k\ (\lambda x_n.(\mathsf{setref!}_k\ (\lambda d. \cdots\ (\mathsf{setref!}_k\ (\lambda d.P) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_n\ W_n)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_1\ W_1)) \\
\qquad\qquad\qquad\quad 0)) \\
\quad 0)
\end{array}$$

where d is not free in any of the x_i, W_i or P.

The CPS language includes a constant O_k for every functional constant O in the source language. The main difference between O_k and O is that the former takes an additional continuation argument that receives the result of the primitive application (if any). Thus, $(O_k\ K\ W)$ is essentially equivalent to $(K\ (O\ W))$. We do not use the latter term because, contrary to the spirit of CPS translation (cf. Theorem 1), its evaluation is sensitive to the parameter-passing technique. For example, the evaluation of $((\lambda d.\ulcorner 8 \urcorner)\ (/\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner))$ yields an error $(\ulcorner 5 \urcorner)$ under call-by-value and yields $\ulcorner 8 \urcorner$ under call-by-name. Similarly, assuming that x is a reference cell whose contents is $\ulcorner 1 \urcorner$, the evaluation of the term:

$$\rho\{(x, \ulcorner 1 \urcorner)\}.((\lambda d.(\mathsf{deref}\ x))\ (\mathsf{setref!}\ x\ \ulcorner 2 \urcorner))$$

yields $\ulcorner 2 \urcorner$ under call-by-value and yields $\ulcorner 1 \urcorner$ under call-by-name.

$$
\begin{array}{rcll}
(\textbf{if}\ \mathsf{true}\ P\ Q) & = & P & (\mathit{Ifk}_t) \\
(\textbf{if}\ \mathsf{false}\ P\ Q) & = & Q & (\mathit{Ifk}_f) \\
\\
(\mathsf{add1}_k\ K\ \ulcorner n \urcorner) & = & (K\ \ulcorner n+1 \urcorner) & (\mathit{Addk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & (K\ \ulcorner n/m \urcorner) \qquad m \neq 0 & (\mathit{Divk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & \ulcorner 5 \urcorner & (\mathit{Divk}_e) \\
\\
(\mathsf{integer?}_k\ K\ \ulcorner n \urcorner) & = & (K\ \mathsf{true}) & (\mathit{Intk}_t) \\
(\mathsf{integer?}_k\ K\ W) & = & (K\ \mathsf{false}) \quad W \neq \ulcorner n \urcorner & (\mathit{Intf}_t) \\
\\
(\mathsf{ref}_k\ K\ W) & = & \rho\{(x, W)\}.(K\ x) & (\mathit{refk}) \\
 & & x \notin FV(W) & \\
\rho\theta \cup \{(x, W)\}.(\mathsf{deref}_k\ K\ x) & = & \rho\theta \cup \{(x, W)\}.(K\ W) & (\mathit{derefk}) \\
\rho\theta \cup \{(x, W_1)\}.(\mathsf{setref!}_k\ K\ x\ W_2) & = & \rho\theta \cup \{(x, W_2)\}.(K\ W_2) & (\mathit{setrefk}) \\
\rho\theta_1.\rho\theta_2.P & = & \rho\theta_1 \cup \theta_2.P & (\mathit{refk}_\cup)
\end{array}
$$

Figure 6: Additional Axioms F_k for CS_k.

8.3. Equational Correspondence

After setting the basic framework, we can define two translations, a CPS transformation and an inverse, between the languages CS and CS_k.

The CPS transformation of the new constructs in CS is the following:

$$
\begin{array}{rcl}
\mathcal{C}_k[\![E[(\textbf{if}\ V\ M\ N)]]\!] & = & (\textbf{if}\ \Phi[\![V]\!]\ \mathcal{C}_k[\![E[M]]\!]\ \mathcal{C}_k[\![E[N]]\!]) \\
\mathcal{C}_k[\![E[(O\ V)]]\!] & = & (O_k\ \mathcal{K}_k[\![E]\!]\ \Phi[\![V]\!]) \\
\mathcal{C}_k[\![E[(O\ V_1\ V_2)]]\!] & = & (O_k\ \mathcal{K}_k[\![E]\!]\ \Phi[\![V_1]\!]\ \Phi[\![V_2]\!])
\end{array}
$$

$$
\Phi[\![c]\!] = c
$$

$$
\begin{array}{rcl}
\mathcal{K}_k[\![E[(\textbf{if}\ [\ ]\ M\ N)]]\!] & = & (\lambda u.\mathcal{C}_k[\![E[(\textbf{if}\ u\ M\ N)]]\!]) \\
\mathcal{K}_k[\![E[(O\ [\ ])]]\!] & = & (\lambda u.\mathcal{C}_k[\![E[(O\ u)]]\!]) \\
\mathcal{K}_k[\![E[(O\ [\ ]\ M)]]\!] & = & (\lambda u.\mathcal{C}_k[\![E[(O\ u\ M)]]\!]) \\
\mathcal{K}_k[\![E[(O\ V\ [\ ])]]\!] & = & (\lambda u.\mathcal{C}_k[\![E[(O\ V\ u)]]\!])
\end{array}
$$

The clause for conditionals duplicates the evaluation context E in both branches of the conditional expression. This duplication is due to two factors:[13]

1. First, the Fischer CPS transformation for conditionals duplicates the continuation variable k:
$$\mathcal{F}[\![(\mathbf{if}\ M\ N\ L)]\!] = \lambda k.\mathcal{F}[\![M]\!]\ (\lambda u.(\mathbf{if}\ u\ \mathcal{F}[\![N]\!]k\ \mathcal{F}[\![L]\!]k)).$$
2. Second, the transformation $\mathcal{C}_k$ eliminates all the administrative redexes from the output of the Fischer CPS transformation.

The extensions to the CPS language result in simple extensions to the inverse CPS transformation:

$$\begin{aligned}
\mathcal{C}^{-1}[\![(\mathbf{if}\ W\ P_1\ P_2)]\!] &= (\mathbf{if}\ \Phi^{-1}[\![W]\!]\ \mathcal{C}^{-1}[\![P_1]\!]\ \mathcal{C}^{-1}[\![P_2]\!]) \\
\mathcal{C}^{-1}[\![(O_k\ K\ W)]\!] &= \mathcal{K}^{-1}[\![K]\!][(O\ \Phi^{-1}[\![W]\!])] \\
\mathcal{C}^{-1}[\![(O_k\ K\ W_1\ W_2)]\!] &= \mathcal{K}^{-1}[\![K]\!][(O\ \Phi^{-1}[\![W_1]\!]\ \Phi^{-1}[\![W_2]\!])] \\
\\
\Phi^{-1}[\![c]\!] &= c
\end{aligned}$$

In order to establish the correspondence between the source and CPS calculi, we need to prove results similar to Lemma 17, Lemma 18, Lemma 19, and Lemma 20. The previous results cannot be extended immediately since we must eliminate η_v from the call-by-value equational theory. Furthermore, both the source and CPS equational theories include additional axioms for constants and conditionals (F and F_k respectively).

By revisiting the proofs of the four lemmas, we establish the following:

- The proof of Lemma 17 does not rely on η_w.
- The proof of Lemma 18 only uses a restricted version of η_v on continuations, $\mathcal{A}$, and *callcc*:

$$\begin{aligned}
(\mathit{callcc}\ (\lambda k.C[(\lambda x.kx)])) &= (\mathit{callcc}\ (\lambda k.C[k])) && (\eta_{v1}) \\
& \quad\ k \notin \mathit{trap}(C) \\
(\lambda x.\mathit{callcc}\ \lambda k.xk) &= \mathit{callcc} && (\eta_{v2}) \\
(\lambda x.\mathcal{A}x) &= \mathcal{A} && (\eta_{v3})
\end{aligned}$$

[13] Alternative translations that do not cause this exponential increase in the size of the code are:

$$\begin{aligned}
\mathcal{C}_k[\![E[(\mathbf{if}\ V\ M\ N)]]\!] &= ((\lambda k.(\mathbf{if}\ \Phi[\![V]\!]\ \mathcal{C}_k[\![M]\!]\ \mathcal{C}_k[\![N]\!]))\ \mathcal{K}_k[\![E]\!]) \\
\text{or} &\quad ((\mathbf{if}\ \Phi[\![V]\!]\ (\lambda k.\mathcal{C}_k[\![M]\!])\ (\lambda k.\mathcal{C}_k[\![N]\!]))\ \mathcal{K}_k[\![E]\!]).
\end{aligned}$$

The first translation is used by compilers [46] but also duplicates the entire evaluation context once we close the language under $\beta\eta$-reductions. The second translation relies on Allison's [1] CPS translation that keeps the local transfer of control independent of the continuation.

The proof also requires the introduction of the following axiom for conditionals:

$$E[(\textbf{if } M\ N\ L)] \;=\; (\textbf{if } M\ E[N]\ E[L]) \qquad (\mathit{If}_{\mathit{lift}})$$

The axiom is introduced by the compacting phase of the CPS transformation and is thus an administrative call-by-value reduction (cf. Definition 7).

- The proof of Lemma 19 shows that η_v-reductions occur in the source language only as a result of η_w-reductions on CPS terms.
- The proof of Lemma 20 shows that η_w-reductions on CPS terms occur only as a result of η_v-reductions on source terms.

As a consequence, the complete equational theory for Core Scheme $\boldsymbol{\lambda}CS$ with respect to its CPS language consists of all the previously derived axioms except η_v; the corresponding CPS equational theory consists of the axioms $\beta\eta F$ excluding η_w.

9. Summary of the Results

The full equational theories for CS and CS_k are in Figures 7 and 8. The correspondences between the different axiom systems for the sublanguages in the previous sections are summarized in the following table.

Language	Call-by-Value Theory	CPS Theory	η_v/η_w	Typed
Λ	X^-	$\beta\eta_k$	$\surd$	$\surd$
$\Lambda + \mathit{callcc} + \mathcal{A}$	X^-C'	$\beta\eta_k$	$\surd$	$\surd$
CS	$X^-C'F^+$	$\beta\eta_k F_k$		
CS^T	$X^-C'F^+$	$\beta\eta_k F_k$	$\surd$	

The Axioms X^-:

$$
\begin{array}{rcll}
((\lambda x.M)\ V) & = & M[x := V] & (\beta_v) \\
((\lambda x.E[x])\ M) & = & E[M] \qquad x \notin FV(E) & (\beta'_\Omega) \\
E[((\lambda x.M)\ N)] & = & ((\lambda x.E[M])\ N) \qquad x \notin FV(E) & (\beta_{lift})
\end{array}
$$

The Axioms C':

$$
\begin{array}{rcll}
E[(callcc\ M)] & = & callcc\ \lambda k.E[(M\ (\lambda f.(k\ E[f])))] & (C_{lift}) \\
 & & k, f \notin FV(E, M) & \\
(callcc\ \lambda k.((\lambda x.M)\ N)) & = & ((\lambda x.(callcc\ \lambda k.M))\ N) & (C_{tail}) \\
 & & k \notin FV(N) & \\
(callcc\ \lambda k.kM) & = & (callcc\ \lambda k.M) & (C_{current}) \\
(callcc\ \lambda d.M) & = & M \qquad d \notin FV(M) & (C_{elim}) \\
callcc\ \lambda k.C[E[(k\ M)]] & = & callcc\ \lambda k.C[(k\ M)] & (C_{abort}) \\
 & & k \notin trap(C) & \\
E[(\mathcal{A}\ M)] & = & (\mathcal{A}\ M) & (Abort) \\
(callcc\ (\lambda k.C[(\lambda x.kx)])) & = & (callcc\ (\lambda k.C[k])) & (\eta_{v\,1}) \\
(\lambda x.callcc\ \lambda k.xk) & = & callcc & (\eta_{v\,2}) \\
(\lambda x.\mathcal{A}x) & = & \mathcal{A} & (\eta_{v\,3})
\end{array}
$$

The Axioms F^+:

$$
\begin{array}{rcll}
(\textbf{if}\ \mathsf{true}\ M\ N) & = & M & (If_t) \\
(\textbf{if}\ \mathsf{false}\ M\ N) & = & N & (If_f) \\
E[(\textbf{if}\ M\ N\ L)] & = & (\textbf{if}\ M\ E[N]\ E[L]) & (If_{lift}) \\
(\mathsf{add1}\ \ulcorner n \urcorner) & = & \ulcorner n+1 \urcorner & (Add) \\
(/\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & \ulcorner n/m \urcorner \qquad m \neq 0 & (Div) \\
(/\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & \ulcorner 5 \urcorner & (Div_e) \\
(\mathsf{integer?}\ \ulcorner n \urcorner) & = & \mathsf{true} & (Int_t) \\
(\mathsf{integer?}\ V) & = & \mathsf{false} \qquad V \neq \ulcorner n \urcorner & (Int_f) \\
(\mathsf{ref}\ V) & = & \rho\{(x, V)\}.x \qquad x \notin FV(V) & (ref) \\
\rho\theta \cup \{(x, V)\}.E[(\mathsf{deref}\ x)] & = & \rho\theta \cup \{(x, V)\}.E[V] & (deref) \\
\rho\theta \cup \{(x, V_1)\}.E[(\mathsf{setref!}\ x\ V_2)] & = & \rho\theta \cup \{(x, V_2)\}.E[V_2] & (setref) \\
\rho\theta_1.\rho\theta_2.M & = & \rho\theta_1 \cup \theta_2.M & (ref_\cup)
\end{array}
$$

Figure 7: The Theory λCS.

The left column gives the name of the source language. The language CS^T is a simply typed variant of CS. The next two columns list the call-by-value axioms and the corresponding CPS axioms. The names of the axioms refer to the definitions in Figures 7 and 8. The column η_v/η_w includes a check mark if it is possible to extend the theories with η_v and η_w, respectively. The rightmost column includes a check mark if the correspondence holds for the simply typed variant of the language.

The Axioms $\beta\eta_k$:

$$
\begin{array}{rcll}
((\lambda x.P)\ W) & = & P[x := W] & (\beta_w) \\
((\lambda k.K_1)\ K_2) & = & K_1[k := K_2] & (\beta_k) \\
(\lambda x.Kx) & = & K \qquad x \notin FV(K) & (\eta_k)
\end{array}
$$

The Axioms F_k:

$$
\begin{array}{rcll}
(\textbf{if}\ \mathsf{true}\ P\ Q) & = & P & (\mathit{Ifk}_t) \\
(\textbf{if}\ \mathsf{false}\ P\ Q) & = & Q & (\mathit{Ifk}_f) \\
(\mathsf{add1}_k\ K\ \ulcorner n \urcorner) & = & (K\ \ulcorner n+1 \urcorner) \quad n \in \mathbb{N} & (\mathit{Addk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & (K\ \ulcorner n/m \urcorner) m \neq 0 & (\mathit{Divk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & \ulcorner 5 \urcorner & (\mathit{Divk}_e) \\
(\mathsf{integer?}_k\ K\ \ulcorner n \urcorner) & = & (K\ \mathsf{true}) & (\mathit{Intk}_t) \\
(\mathsf{integer?}_k\ K\ W) & = & (K\ \mathsf{false}) \quad W \neq \ulcorner n \urcorner & (\mathit{Intk}_f) \\
(\mathsf{ref}_k\ K\ W) & = & \rho\{(x, W)\}.(K\ x) & (\mathit{refk}) \\
 & & x \notin FV(W) & \\
\rho\theta \cup \{(x, W)\}.(\mathsf{deref}_k\ K\ x) & = & \rho\theta \cup \{(x, W)\}.(K\ W) & (\mathit{derefk}) \\
\rho\theta \cup \{(x, W_1)\}.(\mathsf{setref!}_k\ K\ x\ W_2)] & = & \rho\theta \cup \{(x, W_2)\}.(K\ W_2) & (\mathit{setrefk}) \\
\rho\theta_1.\rho\theta_2.P & = & \rho\theta_1 \cup \theta_2.P & (\mathit{refk}_\cup)
\end{array}
$$

Figure 8: The Theory λCS_k.

For the simply typed languages, our calculi are also "semantically complete" with respect to denotational CPS models [35]. The result is a consequence of the completeness of the λ-calculus with respect to the full type structure [42].

10. Example: Coroutines from Continuations

The equational theory of Core Scheme $\boldsymbol{\lambda}CS$ provides a basis for the semantic manipulation of programs by programmers and programming tools alike. For example, programmers may use the theory to evaluate programs in a symbolic manner, to prove the equivalence of two programs, or to simplify a program by a series of meaning-preserving transformations. The first subsection includes an intuitive explanation of a program that implements coroutines using first-class continuations and the second subsection includes a simplification phase based on the CS axioms.

10.1. The Original Program

For convenience, we use a superset of Core Scheme that includes assignments to variables via **set!**, and various other syntactic extensions [40]:

$$
\begin{aligned}
\mathit{error} &\stackrel{df}{=} \mathcal{A} \\
(\mathbf{let}\ ([x\ M])\ N) &\stackrel{df}{=} ((\lambda x.N)\ M) \\
(\mathbf{begin}\ M\ N) &\stackrel{df}{=} ((\lambda d.N)\ M) \qquad \text{where } d \notin FV(N)
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.MN) &\stackrel{df}{=} (\lambda x.(\mathbf{begin}\ M\ N)) \\
(\mathbf{letrec}\ ([x\ (\lambda y.M)])\ N) &\stackrel{df}{=} (\mathbf{let}\ ([x\ \text{'any}]) \\
&\qquad (\mathbf{begin}\ (\mathbf{set!}\ x\ (\lambda y.M))\ N))
\end{aligned}
$$

The original definition of coroutines using first-class continuations is the following [27]:

```
(define make-coroutine
  (lambda (f)
    (callcc (lambda (maker)
              (let ([LCS 'any])
                (let ([resume (lambda (dest val)
                                (callcc (lambda (k)
                                          (set! LCS k)
                                          (dest val))))])
                  (f resume (resume maker (lambda (v) (LCS v))))
                  (error "fell off end")))))))
```

Intuitively, the procedure *make-coroutine* accepts an argument that contains the programmer's coroutine code. For example, the pseudocode in Figure 9 implements one player in a hypothetical game.

The procedure *resume* handles the transfer of control from one coroutine to the other. As the definition of *make-coroutine* shows, *resume* takes two arguments: a destination that denotes the coroutine to be resumed and a value to be passed to the resumed coroutine. Before actually resuming the destination, *resume* saves the current continuation in the local control state *LCS* of the active coroutine, which makes it possible to resume the current coroutine later in the execution.

10.2. Simplifying the Program

The transformation of the program proceeds by applying one of the axioms of the $\boldsymbol{\lambda}CS$-calculus at each step. For clarity, the redex is surrounded

```
(define Player-1-Code
  (let ([Board (make-board)])
    (lambda (resume his-first-shot)
      (letrec ([loop (lambda (his-shot)
                       (if (his-shot-is-fatal?)
                           (I-lost-the-game)
                           (loop (resume Player-2 (compute-my-shot)))))])
        (loop his-first-shot)))))

(define Player-1 (make-coroutine Player-1-Code))
```

Figure 9: Pseudo Coroutine Code.

by a box. During the transformation, **set!** is treated as a free variable. Alternatively, we could use the axioms of the λ_v-S-calculus [17], but for this example, they are superfluous.

```
(define make-coroutine
  (lambda (f)
    ┌──────────────────────────────────────────────────┐
    │(callcc (lambda (maker)                           │
    │          (let ([LCS 'any])                       │
    │            (let ([resume (lambda (dest val)      │
    │                            (callcc (lambda (k)   │
    │                                      (set! LCS k)│
    │                                      (dest val))))])│
    │              (f resume (resume maker             │
    │                          (lambda (v) (LCS v))))  │
    │              (error "fell off end")))))          │))
    └──────────────────────────────────────────────────┘
```

by two uses of (C_{tail})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (f resume (resume maker
                                (lambda (v) (LCS v))))
              (error "fell off end"))))))))
```

by three uses of (β_v)

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (f resume
                 (callcc (lambda (k)
                           (set! LCS k)
                           (maker (lambda (v) (LCS v))))))
              (error "fell off end")))))))
```

by (C_{lift})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (callcc
                (lambda (kk)
                  (f resume ((lambda (k)
                               (set! LCS k)
                               (maker (lambda (v) (LCS v))))
                             (lambda (x)
                               (kk (begin (f resume x)
                                          (error "fell off end"))) )))
                  (error "fell off end")))))))))
```

by (β_{lift}) applied to the expansion of **begin**

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (callcc
                (lambda (kk)
                  (f resume ((lambda (k)
                               (set! LCS k)
                               (maker (lambda (v) (LCS v))))
                             (lambda (x)
                               (begin (f resume x)
                                      (kk (error "fell off end")) ))))
                  (error "fell off end")))))))))
```

by (*Abort*)

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (callcc
                (lambda (kk)
                  (f resume
                     ((lambda (k)
                        (set! LCS k)
                        (maker (lambda (v) (LCS v))))
                      (lambda (x)
                        (begin (f resume x)
                               (error "fell off end")))))
                  (error "fell off end")))          ))))))
```

by (C_{elim})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc (lambda (maker)
                    (f resume
                       ((lambda (k)
                          (set! LCS k)
                          (maker (lambda (v) (LCS v))))
                        (lambda (x)
                          (begin (f resume x)
                                 (error "fell off end"))))   )))))))
```

by (β_v)

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (f resume
                 (begin
                   (set! LCS
                     (lambda (x)
                       (begin (f resume x)
                              (error "fell off end"))))
                   (maker (lambda (v) (LCS v))))   )))))))
```

by (β_{lift})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (f resume
                 (maker
                   (begin
                     (set! LCS
                       (lambda (x)
                         (begin (f resume x)
                                (error "fell off end"))))
                     (lambda (v) (LCS v)))))          ))))))
```

by (C_{abort})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (callcc
            (lambda (maker)
              (maker
                (begin (set! LCS
                             (lambda (x)
                               (begin (f resume x)
                                      (error "fell off end"))))
                       (lambda (v) (LCS v))))))     ))))
```

by $(C_{current})$ and (C_{elim})

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (let ([resume (lambda (dest val)
                        (callcc (lambda (k)
                                  (set! LCS k)
                                  (dest val))))])
          (begin (set! LCS (lambda (x)
                             (begin (f resume x)
                                    (error "fell off end"))))
                 (lambda (v) (LCS v))))     )))
```

by (β_v)

```
= (define make-coroutine
    (lambda (f)
      (let ([LCS 'any])
        (begin (set! LCS (lambda (x)
                           (begin (f (lambda (dest val)
                                       (callcc
                                         (lambda (k)
                                           (set! LCS k)
                                           (dest val))))
                                     x)
                                  (error "fell off end"))))
               (lambda (v) (LCS v))))))
```

by the **letrec** macro definition

```
= (define make-coroutine
    (lambda (f)
      (letrec ([LCS (lambda (x)
                      (begin (f (lambda (dest val)
                                  (callcc (lambda (k)
                                            (set! LCS k)
                                            (dest val))))
                                x)
                             (error "fell off end")))])
        (lambda (v) (LCS v)))))
```

11. Conclusion and Future Research

The Equational Correspondence Theorems establish that equational reasoning about call-by-value programs can be as powerful as reasoning about their CPS counterparts.

Consequently, any CPS-based programming tool that performs sequences of $\beta\eta$-reductions can be substituted by an equivalent tool that does not require an explicit conversion to CPS. The applications to compilers, partial evaluators, and other transformation systems are numerous. For example, direct compilers like Chez Scheme [28] or Zinc [32] can benefit by including the reductions CS in their basic repertoire of optimizations. Similarly, partial evaluators that use transformations like β_{lift} and β'_{Ω} produce residual programs of better quality [7]. Finally, the set of axioms CS could be the base of an expression simplifier (for compilers and other tools) extending that of Galbiati and Talcott [25].

Our result questions the practice of transforming programs to CPS in order to simplify and improve code generators, partial evaluators, data flow analyzers, and other tools. In fact, we have established that the code generators of typical CPS compilers perform an implicit inverse CPS transformation, and that the true intermediate representation of a typical CPS compiler is the subset of source terms in $\beta_{lift}\beta_{flat}$-normal form [22]. Furthermore, although the literature contains claims that the CPS framework improves the accuracy of data flow analyzers [38], we conjecture that the observed improvements are orthogonal to the "passing of continuations" but are rather side-effects of the axioms CS. We are currently investigating the relationship between the CPS transformation and the precision of data flow analysis.

Acknowledgments

We thank Carolyn Talcott for her outstanding efforts on behalf of this paper, which go far beyond the ordinary duties of an editor. The Scheme code was typeset using Dorai Sitaram's SLaTeX. The referees's comments improved the presentation of the results.

A. Proofs

Proof of the auxiliary claims in Proposition 3 (Page 17). Let $M \in \Lambda$:

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

Proof: The proof 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:

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

The extended function satisfies the following property 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 $\mathcal{K}_k$:

- $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ k) = \mathcal{C}_k[\![M]\!]$ and $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![E]\!]\ k) = \mathcal{K}_k[\![E]\!]$;
- $\mathcal{C}_k[\![M]\!]$ and $\mathcal{K}_k[\![E]\!]$ are in $\overline{\beta}\overline{\eta}$-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, $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = ((\overline{\lambda} k.k\mathcal{F}_v[\![V]\!])\ k) = (k\ \mathcal{F}_v[\![V]\!])$. 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 $\overline{\beta}\overline{\eta}$-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 $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![M]\!]\ c) = \mathcal{C}_c[\![M]\!]$, and $\mathcal{C}_c[\![M]\!]$ is in $\overline{\beta}\overline{\eta}$-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,

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

   There are two cases:
   (a) $V \notin \mathit{Vars}$: then $|E| < |G|$. By the inductive hypothesis, $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash \mathcal{F}[\![E]\!]k = \mathcal{K}_k[\![E]\!]$, and $\mathcal{K}_k[\![E]\!]$ is in $\overline{\beta}\overline{\eta}$-normal form. By an argument similar to case 1, $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash \mathcal{F}_v[\![V]\!] = \Phi[\![V]\!]$, and $\Phi[\![V]\!]$ is in $\overline{\beta}\overline{\eta}$-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 inlining the arguments in cases 4 to 7, $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![E]\!]\ k) = \mathcal{K}_k[\![E]\!]$, and $\mathcal{K}_k[\![E]\!]$ is in $\overline{\beta}\overline{\eta}$-normal form. The result follows as in subcase (a).
3. $G = E[((\lambda x.M)\ V)]$: then,

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

   The result follows by the inductive hypothesis and an argument similar to case 1.
4. $G = [\ ]$: then $\boldsymbol{\lambda}\overline{\beta}\overline{\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = ((\overline{\lambda} k.k)\ k) = k = \mathcal{K}_k[\![G]\!]$. Moreover $\mathcal{K}_k[\![G]\!]$ is in $\overline{\beta}\overline{\eta}$-normal form.
5. $G = E[(x\ [\ ])]$: then,

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

   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 $\overline{\beta\eta}$-normal form. Therefore, $\boldsymbol{\lambda}\overline{\beta\eta} \vdash (\mathcal{F}[\![G]\!]\ k) = \mathcal{K}_k[\![G]\!]$. 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 $\overline{\beta\eta}$-normal form.

∎

Proof of the second claim in Lemma 4 (Page 17). Let $M, L \in \Lambda$, $E \in \mathit{EvCont}$, and $z \in \mathit{Vars}$:

$$\begin{aligned}\mathcal{C}_k[\![((z\ M)\ L)]\!] &\equiv \mathcal{C}_k[\![((\lambda u.uL)\ (z\ M))]\!] \quad \text{where } u \notin FV(L) \\ \mathcal{K}_k[\![((z\ E)\ L)]\!] &\equiv \mathcal{K}_k[\![((\lambda u.uL)\ (z\ E))]\!] \quad \text{where } u \notin FV(L)\end{aligned}$$

Proof: The proof is by induction on the size of M and E and proceeds by cases:

1. $M = V$, then $\mathcal{C}_k[\![((z\ V)\ L)]\!]$ is identical to:

$$((z\ \mathcal{K}_k[\![([\,]\ L)]\!])\ \Phi[\![V]\!]) \equiv ((z\ \overline{\lambda} u.\mathcal{C}_k[\![(u\ L)]\!])\ \Phi[\![V]\!]).$$

The right hand side $\mathcal{C}_k[\![((\lambda u.uL)\ (z\ V))]\!]$ is:

$$((z\ \mathcal{K}_k[\![((\lambda u.uL)\ [\,])]\!])\ \Phi[\![V]\!]) \equiv ((z\ (\lambda u.\mathcal{C}_k[\![(uL)]\!]))\ \Phi[\![V]\!]).$$

The two sides of the equation differ by the overline above the abstraction λu. The terms are identical because the abstraction λu cannot be part of an administrative $\overline{\eta}$ redex in the first term.
2. $M = E[(y\ V)]$, then $\mathcal{C}_k[\![((z\ E[(y\ V)])\ L)]\!]$

$$\begin{aligned}&\equiv ((y\ \mathcal{K}_k[\![((z\ E)\ L)]\!])\ \Phi[\![V]\!]) \\ &\equiv ((y\ \mathcal{K}_k[\![((\lambda u.uL)\ (z\ E))]\!])\ \Phi[\![V]\!]) \quad \text{(induction)}\end{aligned}$$

3. $M = E[((\lambda x.N)\ V)]$, then $\mathcal{C}_k[\![((z\ E[((\lambda x.N)\ V)])\ L)]\!]$

$$\begin{aligned}&\equiv ((\lambda x.\mathcal{C}_k[\![((z\ E[N])\ L)]\!])\ \Phi[\![V]\!]) \\ &\equiv ((\lambda x.\mathcal{C}_k[\![((\lambda u.uL)\ (z\ E[N]))]\!])\ \Phi[\![V]\!]) \quad \text{(induction)}\end{aligned}$$

4. $E = [\,]$, then $\mathcal{K}_k[\![((z\ [\,])\ L)]\!] \equiv (z\ \mathcal{K}_k[\![([\,]\ L)]\!]) \equiv (z\ \overline{\lambda} u.\mathcal{C}_k[\![(uL)]\!])$. The right hand side is $(z\ \lambda u.\mathcal{C}_k[\![(uL)]\!])$. The result follows because the overline does not create an administrative redex.
5. $E = E_1[(x\ [\,])]$, then $\mathcal{K}_k[\![((z\ E_1[(x\ [\,])])\ L)]\!]$

$$\begin{array}{ll} \equiv & (x\ \mathcal{K}_k[\![((z\ E_1)\ L)]\!]) \\ \equiv & (x\ \mathcal{K}_k[\![((\lambda u.uL)\ (z\ E_1))]\!]) \quad \text{(induction)} \end{array}$$

6. $E = E_1[((\lambda x.N)\ [\,])]$, then $\mathcal{K}_k[\![((z\ E_1[((\lambda x.N)\ [\,])])\ L)]\!]$

$$\begin{array}{ll} \equiv & (\lambda x.\mathcal{C}_k[\![((z\ E_1[N])\ L)]\!]) \\ \equiv & (\lambda x.\mathcal{C}_k[\![((\lambda u.uL)\ (z\ E_1[N]))]\!]) \quad \text{(induction)} \end{array}$$

7. $E = E_1[([\,]\ N)]$, then $\mathcal{K}_k[\![((z\ E_1[([\,]\ N)])\ L)]\!]$

$$\begin{array}{ll} \equiv & \overline{\lambda} f.\mathcal{C}_k[\![((z\ E_1[(f\ N)])\ L)]\!] \\ \equiv & \overline{\lambda} f.\mathcal{C}_k[\![((\lambda u.uL)\ (z\ E_1[(f\ N)]))]\!] \quad \text{(induction)} \end{array}$$

∎

Proof of the auxiliary claim in Theorem 5 (Page 19).

- For all $P \in cps(\Lambda)$, there exists $M \in \Lambda$ such that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \twoheadrightarrow P$.
- For all $K \in cps(EvCont)$, there exists $E \in EvCont$ such that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{K}_k[\![E]\!] \twoheadrightarrow K$.

Proof: The proof is by lexicographic induction on $\langle \tilde{G}, |G| \rangle$ where G is an element P of $cps(\Lambda)$ or an element K of $cps(EvCont)$, $\tilde{G}$ is the number of abstractions of the form $\lambda k.K$ in G, and $|G|$ is the size of G. The proof proceeds by case analysis on the possible elements of $cps(\Lambda)$ and $cps(EvCont)$:

1. $G = (k\ W)$: there are four cases:
   (a) $W = x$: take $M = x$.
   (b) $W = \lambda k.k$: take $M = \lambda x.x$.
   (c) $W = \lambda k.W_1 K$: let $P_1 = ((W_1 K)\ x)$, then $\tilde{P_1} < \tilde{G}$ because P_1 has one less abstraction of the form $\lambda k.K$ than G. Therefore, by the inductive hypothesis, there exists an M_1 such that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M_1]\!] \twoheadrightarrow P_1$. Take $M = \lambda x.M_1$.
   (d) $W = \lambda k.\lambda x.P_1$: by the inductive hypothesis, P_1 is reachable from a term M_1. Take $M = \lambda x.M_1$.
2. $G = ((x\ K)\ W)$: by the inductive hypothesis, K is reachable from an evaluation context E. By an argument similar to the first case, W is reachable from a value V. Take $M = E[(x\ V)]$.

3. $G = (((\lambda k.K_1)\ K_2)\ W)$: by the inductive hypothesis, K_2 is reachable from an evaluation context E_2. By repeating the argument for the first case, the values $\lambda k.K_1$ and W are reachable from V_1 and V respectively. Let M be the following term $((\lambda x.((\lambda y.E_2[(y\ x)])\ V_1))\ V)$. Then $\mathcal{C}_k[\![M]\!]$

$$\begin{array}{lll} \equiv & ((\lambda x.((\lambda y.((y\ \mathcal{K}_k[\![E_2]\!])\ x))\ \Phi[\![V_1]\!]))\ \Phi[\![V]\!]) & \\ \twoheadrightarrow & ((\lambda x.((\lambda y.((y\ K_2)\ x))\ \lambda k.K_1))\ W) & \text{(induction)} \\ \longrightarrow & ((\lambda x.(((\lambda k.K_1)\ K_2)\ x))\ W) & (\beta) \\ \longrightarrow & (((\lambda k.K_1)\ K_2)\ W) & (\beta) \end{array}$$

4. $G = ((\lambda x.P_1)\ W)$: by the inductive hypothesis, there exists an M_1 that reaches P_1. By repeating the argument for the first case, there also exists a value V that reaches W. Take $M = ((\lambda x.M_1)\ V)$.
5. $G = k$: take $E = [\,]$.
6. $G = (x\ K_1)$: by the inductive hypothesis, there exists an E_1 that reaches K_1. Take $E = E_1[(x\ [\,])]$.
7. $G = ((\lambda k.K_1)\ K_2)$: similarly to case 3, take E to be the evaluation context $((\lambda x.((\lambda y.E_2[(y\ x)])\ V_1))\ [\,])$.
8. $G = (\lambda x.P_1)$: take $E = ((\lambda x.M_1)\ [\,])$ where M_1 reaches P_1 by induction.

∎

Proof of Lemma 6 (Page 19). Let $P_1 \in cps(\Lambda)$, $W_1 \in cps(\mathit{Values})$ and $K_1 \in cps(\mathit{EvCont})$, then,

1. $\boldsymbol{\lambda}\beta\eta \vdash P_1 \longrightarrow P_2$ implies $P_2 \in cps(\Lambda)$.
2. $\boldsymbol{\lambda}\beta\eta \vdash W_1 \longrightarrow W_2$ implies $W_2 \in cps(\mathit{Values})$.
3. $\boldsymbol{\lambda}\beta\eta \vdash K_1 \longrightarrow K_2$ implies $K_2 \in cps(\mathit{EvCont})$.

Proof: The proof is by induction on the structure of the terms P_1, W_1 and K_1.

1. Let $P_1 \in cps(\Lambda)$ and assume $\boldsymbol{\lambda}\beta\eta \vdash P_1 \longrightarrow P_2$. By definition, P_1 must be of the form $(K_1\ W_1)$ with $K_1 \in cps(\mathit{EvCont})$ and $W_1 \in cps(\mathit{Values})$. Three kinds of reductions are possible:
   - $\boldsymbol{\lambda}\beta\eta \vdash (K_1\ W_1) \longrightarrow (K_2\ W_1)$ because $K_1 \longrightarrow K_2$. By the inductive hypothesis, $K_2 \in cps(\mathit{EvCont})$ and therefore $P_2 \in cps(\Lambda)$.
   - $\boldsymbol{\lambda}\beta\eta \vdash (K_1\ W_1) \longrightarrow (K_1\ W_2)$ because $W_1 \longrightarrow W_2$. The result follows by the inductive hypothesis.

- $\boldsymbol{\lambda}\beta\eta \vdash ((\lambda x.P)\ W_1) \longrightarrow P[x := W_1]$ because $K_1 = (\lambda x.P)$. By a simple inductive argument, the substitution preserves the syntactic category.

2. Let $W_1 \in cps(Values)$ and assume $\boldsymbol{\lambda}\beta\eta \vdash W_1 \longrightarrow W_2$. The term W_1 cannot be a variable, thus $W_1 = \lambda k.K_1$ where $K_1 \in cps(EvCont)$. Either:
   - $\boldsymbol{\lambda}\beta\eta \vdash \lambda k.K_1 \longrightarrow \lambda k.K_2$ because $K_1 \longrightarrow K_2$. The result follows by the inductive hypothesis.
   - $\boldsymbol{\lambda}\beta\eta \vdash \lambda k.W_3 k \longrightarrow W_3$ because $K_1 = (W_3\ k)$ and $W_3 \in cps(Values)$ by definition.
3. Let $K_1 \in cps(EvCont)$ and assume $\boldsymbol{\lambda}\beta\eta \vdash K_1 \longrightarrow K_2$. The term K_1 cannot be a variable, thus there are two cases:
   - $K_1 = \lambda x.P_1$ and there are two subcases:
     - $\boldsymbol{\lambda}\beta\eta \vdash (\lambda x.P_1) \longrightarrow (\lambda x.P_2)$ because $P_1 \longrightarrow P_2$. The result follows by the inductive hypothesis.
     - $\boldsymbol{\lambda}\beta\eta \vdash (\lambda x.Kx) \longrightarrow K$ because $P_1 = Kx$ and $K \in cps(EvCont)$ by definition.
   - $K_1 = (W\ K)$ and there are three cases:
     - $\boldsymbol{\lambda}\beta\eta \vdash (W\ K) \longrightarrow (W_1\ K)$ because $W \longrightarrow W_1$ and the result follows by the inductive hypothesis.
     - $\boldsymbol{\lambda}\beta\eta \vdash (W\ K) \longrightarrow (W\ K_3)$ because $K \longrightarrow K_3$ and the result follows also by induction.
     - $\boldsymbol{\lambda}\beta\eta \vdash ((\lambda k.K_3)\ K) \longrightarrow K_3[k := K]$ because $W = \lambda k.K_3$. By an inductive argument, the substitution preserves the syntactic category.

∎

Proof of Theorem 8 (Page 21). Let $P \in cps(\Lambda)$, $K \in cps(EvCont)$. Then,

1. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] = P$ and $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!] = K$;
2. $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] \equiv P$ and $(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!] \equiv K$ if there exists $M \in \Lambda$ and $E \in EvCont$ such that $P = \mathcal{C}_k[\![M]\!]$ and $K = \mathcal{K}_k[\![E]\!]$.

Proof: The proof of the first claim is by lexicographic induction on the number of abstractions of the form $\lambda k.K$ and the size of the G where G is an element of $cps(\Lambda)$ or $cps(Evcont)$:

1. $G = (k\ W)$, then there are four cases:
   (a) $W = x$: Then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] \equiv (k\ x) \equiv G$.
   (b) $W = \lambda k.k$: Then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] \equiv (k\ \lambda k.\lambda x.kx) = (k\ \lambda k.k)$.

(c) $W = \lambda k.W_1 K$: Then,

$$\begin{array}{rcl}(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] & \equiv & (k\ \lambda k.\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![((W_1\ K)\ x)]\!]) \\ & = & (k\ \lambda k.\lambda x.((W_1\ K)\ x)) \\ & & \quad \text{(induction)} \\ & = & (k\ \lambda k.(W_1\ K)).\end{array}$$

(d) $W = \lambda k.\lambda x.P_1$: Then,

$$(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] \equiv (k\ \lambda k.\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P_1]\!]),$$

and the result follows by the inductive hypothesis.

2. $G = ((x\ K)\ W)$: Then,

$$(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] \equiv ((x\ (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]).$$

By the inductive hypothesis $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!] = K$, and by an argument similar to case 1, $\boldsymbol{\lambda}\beta\eta \vdash (\Phi \circ \Phi^{-1})[\![W]\!] = W$.

3. $G = (((\lambda k.K_1)\ K_2)\ W)$: Then,

$$\begin{array}{rcl}(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] & \equiv & ((\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K_1[k := K_2]]\!]\ (\Phi \circ \Phi^{-1})[\![W]\!]) \\ & = & (K_1[k := K_2]\ (\Phi \circ \Phi^{-1})[\![W]\!]) \\ & & \quad \text{(induction)} \\ & = & (K_1[k := K_2]\ W) \qquad \text{(similar to case 1)} \\ & = & (((\lambda k.K_1)\ K_2)\ W) \qquad (\beta)\end{array}$$

4. $G = ((\lambda x.P_1)\ W)$: Then,

$$\begin{array}{rcl}(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![G]\!] & \equiv & ((\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P_1]\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]) \\ & = & ((\lambda x.P_1)\ (\Phi \circ \Phi^{-1})[\![W]\!]) \qquad \text{(induction)} \\ & = & ((\lambda x.P_1)\ W) \qquad \text{(similar to case 1)}\end{array}$$

5. $G = k$: Then $(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![G]\!] \equiv G$.
6. $G = (x\ K_1)$: Then $(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![G]\!] \equiv (x\ (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K_1]\!])$ and the result follows by the inductive hypothesis.
7. $G = ((\lambda k.K_1)\ K_2)$: Then,

$$(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![G]\!] \equiv (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K_1[k := K_2]]\!],$$

and this case is similar to case 3.

8. $G = (\lambda x.P_1)$: Then $(\mathcal{K}_k \circ \mathcal{K}^{-1})[\![G]\!] \equiv (\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P_1]\!])$ and the result follows by induction.

The proof of the second part is identical to the above but it excludes the cases that do not correspond to images of source terms. In particular, it excludes cases 1b and 1c because, in the image of a source term, the body of a $\lambda k.K$ abstraction must be of the form $\lambda x.P$. Moreover, it excludes cases 3 and 7 because they contain the administrative redex $((\lambda k.K_1)\ K_2)$ and hence are not the image of any source term. ∎

Proof of Lemma 9 (Page 22). Let $P \in cps(\Lambda)$ and $K \in cps(EvCont)$, then, $\mathcal{C}^{-1}[\![P]\!] \in \Lambda_a$ and $\mathcal{K}^{-1}[\![K]\!] \in EvCont_a$.

Proof: The proof is by lexicographic induction on the number of abstractions of the form $\lambda k.K$ and the size of the terms. It proceeds by case analysis on the possible inputs to $\mathcal{C}^{-1}$ and $\mathcal{K}^{-1}$:

1. $P = (K\ W)$, then $\mathcal{C}^{-1}[\![P]\!] \equiv \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]]$. By induction, $\mathcal{K}^{-1}[\![K]\!] \in EvCont_a$. It remains to establish that $\Phi^{-1}[\![W]\!] \in Values_a$.
   (a) $W = x$, then $\Phi^{-1}[\![W]\!] \equiv x \in Values_a$.
   (b) $W = \lambda k.k$, then $\Phi^{-1}[\![W]\!] \equiv \lambda x.x \in Values_a$.
   (c) $W = \lambda k.WK$, then $\Phi^{-1}[\![W]\!] \equiv \lambda x.\mathcal{C}^{-1}[\![((WK)\ x)]\!]$. Because the term $((WK)\ x)$ has one less abstraction of the form $\lambda k.K$ than W, the inductive hypothesis applies to it. Therefore $\mathcal{C}^{-1}[\![((WK)\ x)]\!] \in \Lambda_a$ which shows that the term $\Phi^{-1}[\![W]\!] \in Values_a$.
   (d) $W = \lambda k.\lambda x.P$, then $\Phi^{-1}[\![W]\!] \equiv \lambda x.\mathcal{C}^{-1}[\![P]\!]$, and the result follows by induction.
2. $K = k$, then $\mathcal{K}^{-1}[\![K]\!] \equiv [\ ] \in EvCont_a$.
3. $K = (x\ K_1)$, then $\mathcal{K}^{-1}[\![K]\!] \equiv \mathcal{K}^{-1}[\![K_1]\!][(x\ [\ ])]$. The result is immediate because $\mathcal{K}^{-1}[\![K_1]\!] \in EvCont_a$ by induction.
4. $K = ((\lambda k.K_1)\ K_2)$, then $\mathcal{K}^{-1}[\![K]\!] \equiv \mathcal{K}^{-1}[\![K_1[k := K_2]]\!]$. Because k occurs exactly once in K_1, then term $K_1[k := K_2]$ has one less abstraction of the $\lambda k.K$ than $((\lambda k.K_1)\ K_2)$. Therefore, $\mathcal{K}^{-1}[\![K]\!] \in EvCont_a$ by the inductive hypothesis.
5. $K = \lambda x.P$, then $\mathcal{K}^{-1}[\![K]\!] \equiv ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\ ])$ and the result follows by induction.

∎

Proof of Theorem 10 (Page 22). Let $M \in \Lambda$, then:

1. $\boldsymbol{\lambda}\beta_{lift}\beta_{flat} \vdash M \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$,
2. $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv M$ if there exists $P \in cps(\Lambda)$ such that $M = \mathcal{C}^{-1}[\![P]\!]$.

Proof: In order to prove the first claim, we strengthen the statement as follows.

$$\begin{aligned} \boldsymbol{\lambda}\beta_{lift}\beta_{flat} &\vdash M \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \text{ and} \\ \boldsymbol{\lambda}\beta_{lift}\beta_{flat} &\vdash E[(x\ L)] \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!][(x\ L)] \end{aligned}$$

The proof is by induction on the size of M or E and proceeds by cases:

1. $M = V$: then there are two subcases:
   (a) $V = x$: then $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![x]\!] \equiv x$.
   (b) $V = \lambda x.N$: then,

$$\begin{array}{lll} \lambda x.N & \twoheadrightarrow & \lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![N]\!] \qquad \text{(induction)} \\ & \equiv & (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![\lambda x.N]\!] \end{array}$$

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

$$(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!][(x\ (\Phi^{-1} \circ \Phi)[\![V]\!])].$$

   There are two cases:
   (a) $V \notin Vars$, then $|E| < |E[(x\ V)]|$ and the inductive hypothesis applies, *i.e.*, $E[(x\ V)] \twoheadrightarrow (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!][(x\ V)]$. The result follows because $V \twoheadrightarrow (\Phi^{-1} \circ \Phi)[\![V]\!]$ as in case 1.
   (b) $V = y \in Vars$, then there are four cases depending on the structure of E:
      i. $M = (x\ y)$, then $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv M$.
      ii. $M = E_1[(z\ (x\ y))]$, then the result follows by induction.
      iii. $M = E_1[((\lambda z.L)\ (x\ y))]$, then,

$$\begin{array}{lll} M & \longrightarrow & ((\lambda z.E_1[L])\ (x\ y)) \qquad (\beta_{lift}) \\ & \twoheadrightarrow & ((\lambda z.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[L]]\!])\ (x\ y)) \\ & \equiv & (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]. \end{array}$$

      iv. $M = E_1[((x\ y)\ L)]$, then

$$\begin{array}{lll} M & \longrightarrow & E_1[((\lambda u.(u\ L))\ (x\ y))] \qquad (\beta_{flat}) \\ & \longrightarrow & ((\lambda u.E_1[(u\ L)])\ (x\ y)) \qquad (\beta_{lift}) \\ & \twoheadrightarrow & ((\lambda u.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[(u\ L)]]\!])\ (x\ y)) \\ & \equiv & (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]. \end{array}$$

3. $M = E_1[((\lambda x.N)\ V)]$: then

$$(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] \equiv ((\lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[N]]\!])\ (\Phi^{-1} \circ \Phi)[\![V]\!]).$$

   The left hand side M:

$$\begin{array}{ll} \longrightarrow & ((\lambda x.E_1[N])\ V) \qquad (\beta_{lift}) \\ \twoheadrightarrow & ((\lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[N]]\!])\ V) \\ \twoheadrightarrow & ((\lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[N]]\!])\ (\Phi^{-1} \circ \Phi)[\![V]\!]). \end{array}$$

4. $E = [\,]$: then $(\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![[\,]]\!][(x\ L)] \equiv (x\ L)$.
5. $E = E_1[(z\ [\,])]$: then we want to show that $E_1[(z\ (x\ L))] \twoheadrightarrow (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[(z\ [\,])]]\!][(x\ L)] \equiv (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1]\!][(z\ (x\ L))]$. The result is immediate since the inductive hypothesis applies to E_1.

6. $E = E_1[((\lambda z.N)\ [\ ])]$: then, $E_1[((\lambda z.N)\ (x\ L))]$

$$\begin{array}{lll} \longrightarrow\!\!\!\!\rightarrow & ((\lambda z.E_1[N])\ (x\ L)) & (\beta_{lift}) \\ \longrightarrow\!\!\!\!\rightarrow & ((\lambda z.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[N]]\!])\ (x\ L)) & \text{(induction)} \\ \longrightarrow\!\!\!\!\rightarrow & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[((\lambda z.N)\ [\ ])]]\!][(x\ L)]. & \end{array}$$

7. $E = E_1[([\ ]\ N)]$: then, $E_1[((x\ L)\ N)]$

$$\begin{array}{lll} \longrightarrow\!\!\!\!\rightarrow & E_1[((\lambda u.uN)\ (x\ L))] & (\beta_{flat}) \\ \longrightarrow\!\!\!\!\rightarrow & ((\lambda u.E_1[(u\ N)])\ (x\ L)) & (\beta_{lift}) \\ \longrightarrow\!\!\!\!\rightarrow & ((\lambda u.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[(u\ N)]]\!])\ (x\ L)) & \text{(induction)} \\ \longrightarrow\!\!\!\!\rightarrow & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[([\ ]\ N)]]\!][(x\ L)]. & \end{array}$$

The proof of the second claim proceeds as above but is restricted to terms of the form $\mathcal{C}^{-1}[\![P]\!]$ for some $P \in cps(\Lambda)$. By the grammar in Definition 10, the context E_1 in cases 2b(iii), 3, and 6 must be empty. Also cases 2b(iv) and 7 are impossible. Since these cases account for all the reductions, it follows that M is identical to $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$. ∎

Proof of case 4 in Lemma 12 (Page 24). Let $P \in cps(\Lambda)$, and $W \in cps(Values)$, then:

$$\boldsymbol{\lambda}\beta_v\beta_{lift}\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![((\lambda x.P)\ W)]\!] \longrightarrow\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![P[x := W]]\!].$$

Proof:

$$\begin{array}{lll} \boldsymbol{\lambda}\beta_v\beta_{lift}\beta_{id}\beta_\Omega \vdash & & \mathcal{C}^{-1}[\![((\lambda x.P)\ W)]\!] \\ & \equiv & ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ \Phi^{-1}[\![W]\!]) \\ & \longrightarrow & \mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]] \\ & \longrightarrow\!\!\!\!\rightarrow & \mathcal{C}^{-1}[\![P[x := W]]\!] \end{array}$$

The first three steps are straightforward; the last step requires an appropriate proof. We prove the following claims:

1. $\boldsymbol{\lambda}\beta_v\beta_{lift}\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]] \longrightarrow\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![P[x := W]]\!]$.
2. $\boldsymbol{\lambda}\beta_v\beta_{lift}\beta_{id}\beta_\Omega \vdash \mathcal{K}^{-1}[\![K]\!][x := \Phi^{-1}[\![W]\!]] \longrightarrow\!\!\!\!\rightarrow \mathcal{K}^{-1}[\![K[x := W]]\!]$.

The proof relies on an auxiliary claim that we state and prove after the main proof. The main proof is by lexicographic induction on $\langle \tilde{G}, |G| \rangle$, *i.e.*, the number of abstractions of the form $\lambda k.K$ and the size of the terms. The proof proceeds by cases on the arguments to $\mathcal{C}^{-1}$ and $\mathcal{K}^{-1}$:

1. $P = (K\ W_1)$: then $\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{ll} \equiv & \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W_1]\!]]\ [x := \Phi^{-1}[\![W]\!]] \\ \longrightarrow\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K[x := W]]\!][\Phi^{-1}[\![W_1]\!][x := \Phi^{-1}[\![W]\!]]]. \end{array}$$

It remains to establish that substitution commutes with Φ^{-1} as well. There are five cases:

(a) $W_1 = x$: then $\Phi^{-1}[\![x]\!][x := \Phi^{-1}[\![W]\!]] \equiv \Phi^{-1}[\![x[x := W]]\!]$.
(b) $W_1 = z$ and $z \neq x$: the result is immediate.
(c) $W_1 = \lambda k.k$: immediate since x is not free.
(d) $W_1 = \lambda k.W_2 k$: then $\Phi^{-1}[\![W_1]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{ll} \equiv & \lambda z.\mathcal{C}^{-1}[\![((W_2 K)\ z)]\!][x := \Phi^{-1}[\![W]\!]] \\ \longrightarrow\!\!\!\!\!\rightarrow & \lambda z.\mathcal{C}^{-1}[\![(W_2 K)[x := W]\ z]\!] \\ \equiv & \Phi^{-1}[\![(\lambda k.W_2 K)[x := W]]\!]. \end{array}$$

(e) $W_1 = \lambda k.\lambda z.P_1\ (z \neq x)$: then $\Phi^{-1}[\![W_1]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{lll} \equiv & (\lambda z.\mathcal{C}^{-1}[\![P]\!])[x := \Phi^{-1}[\![W]\!]] & \\ \equiv & (\lambda z.\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]]) & \\ \longrightarrow\!\!\!\!\!\rightarrow & \lambda z.\mathcal{C}^{-1}[\![P[x := W]]\!] & \text{(induction)} \\ \equiv & \Phi^{-1}[\![(\lambda k.\lambda z.P)[x := W]]\!] & \end{array}$$

2. $K = k$: then the claim is vacuously true because $k \neq x$.
3. $K = ((\lambda k.K_1)\ K_2)$: then $\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{ll} \equiv & \mathcal{K}^{-1}[\![K_1[k := K_2]]\!][x := \Phi^{-1}[\![W]\!]] \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K_1[k := K_2][x := W]]\!] \\ \equiv & \mathcal{K}^{-1}[\![((\lambda k.K_1[x := W])\ K_2[x := W])]\!]. \end{array}$$

4. $K = \lambda z.P_1\ (z \neq x)$: then $\mathcal{K}^{-1}[\![\lambda z.P_1]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{ll} \equiv & ((\lambda z.\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]])\ [\,]) \\ \longrightarrow\!\!\!\!\!\rightarrow & ((\lambda z.\mathcal{C}^{-1}[\![P[x := W]]\!])\ [\,]) \\ \equiv & \mathcal{K}^{-1}[\![\lambda z.P[x := W]]\!]. \end{array}$$

5. $K = \lambda x.P_1$: immediate since x is not free.
6. $K = (z\ K_1)$ and $z \neq x$: this is a special case of the next clause.
7. $K = (x\ K_1)$: then $\mathcal{K}^{-1}[\![(x\ K_1)]\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{array}{ll} \equiv & \mathcal{K}^{-1}[\![K_1]\!][(x\ [\,])][x := \Phi^{-1}[\![W]\!]] \\ \equiv & \mathcal{K}^{-1}[\![K_1]\!][x := \Phi^{-1}[\![W]\!]][(\Phi^{-1}[\![W]\!]\ [\,])] \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K_1[x := W]]\!]\ [(\Phi^{-1}[\![W]\!]\ [\,])]. \end{array}$$

For readability, let $K' = K_1[x := W]$. The goal is to prove that:

$$\mathcal{K}^{-1}[\![K']\!][(\Phi^{-1}[\![W]\!]\ [\,])] \text{ reduces to } \mathcal{K}^{-1}[\![(W\ K')]\!].$$

We proceed by cases of W:

(a) $W = y$: then $\mathcal{K}^{-1}[\![K']\!][(y\ [\,])] \equiv \mathcal{K}^{-1}[\![(y\ K')]\!]$.

(b) $W = \lambda k.k$: then,

$$\begin{array}{lll} \mathcal{K}^{-1}[\![K']\!][((\lambda x.x)\ [\ ])] & \longrightarrow & \mathcal{K}^{-1}[\![K']\!] \qquad (\beta_{id}) \\ & \equiv & \mathcal{K}^{-1}[\![((\lambda k.k)\ K')]\!] \end{array}$$

(c) $W = \lambda k.W_3 K$: then, $\mathcal{K}^{-1}[\![K']\!][((\lambda y.\mathcal{C}^{-1}[\![((W_3 K)\ y)]\!])\ [\ ])]$

$$\begin{array}{lll} \equiv & \mathcal{K}^{-1}[\![K']\!][\mathcal{K}^{-1}[\![(\lambda y.((W_3 K)\ y))]\!]] & \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K']\!][\mathcal{K}^{-1}[\![(W_3 K)]\!]] & \text{(case 1 (Lem. 12))} \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![(W_3 K)[k := K']]\!] & \text{(aux. claim)} \\ \equiv & \mathcal{K}^{-1}[\![((\lambda k.W_3 K)\ K')]\!]. & \end{array}$$

(d) $W = \lambda k.\lambda z.P_2$: then $\mathcal{K}^{-1}[\![K']\!][((\lambda z.\mathcal{C}^{-1}[\![P_2]\!])\ [\ ])]$

$$\begin{array}{lll} \longrightarrow & ((\lambda z.\mathcal{K}^{-1}[\![K']\!][\mathcal{C}^{-1}[\![P_2]\!]])\ [\ ]) & (\beta_{lift}) \\ \longrightarrow\!\!\!\!\!\rightarrow & ((\lambda z.\mathcal{C}^{-1}[\![P_2[k := K']]\!])\ [\ ]) & \text{(aux. claim)} \\ \equiv & \mathcal{K}^{-1}[\![\lambda z.P_2[k := K']]\!] & \\ \equiv & \mathcal{K}^{-1}[\![((\lambda k.\lambda z.P_2)\ K')]\!]. & \end{array}$$

Auxiliary Claim:

Let $P \in cps(\Lambda)$, and let $K, K_1, K_2 \in cps(EvCont)$, then,

1. $\boldsymbol{\lambda}\beta_{lift} \vdash \mathcal{K}^{-1}[\![K]\!][\mathcal{C}^{-1}[\![P]\!]] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![P[k := K]]\!]$
2. $\boldsymbol{\lambda}\beta_{lift} \vdash \mathcal{K}^{-1}[\![K_2]\!][\mathcal{K}^{-1}[\![K_1]\!]] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{K}^{-1}[\![K_1[k := K_2]]\!]$

The proof of the auxiliary claim is by induction on the number of abstractions of the $\lambda k.K$ and the size of P or K_1. It proceeds by case analysis on the possible elements of $cps(\Lambda)$ and $cps(EvCont)$:

1. $P = (K_3\ W)$, then $\mathcal{K}^{-1}[\![K]\!][\mathcal{C}^{-1}[\![P]\!]]$

$$\begin{array}{ll} \equiv & \mathcal{K}^{-1}[\![K]\!][\mathcal{K}^{-1}[\![K_3]\!][\Phi^{-1}[\![W]\!]]] \\ \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K_3[k := K]]\!][\Phi^{-1}[\![W]\!]] \\ \equiv & \mathcal{K}^{-1}[\![(K_3\ W)[k := K]]\!]. \end{array}$$

   The last equivalence holds because k is never free in W.
2. $K_1 = k$: then both sides are identical to $\mathcal{K}^{-1}[\![K_2]\!]$.
3. $K_1 = (x\ K_3)$, then $\mathcal{K}^{-1}[\![K_2]\!][\mathcal{K}^{-1}[\![K_3]\!][(x\ [\ ])]]$

$$\begin{array}{ll} \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K_3[k := K_2]]\!][(x\ [\ ])] \\ \equiv & \mathcal{K}^{-1}[\![(x\ K_3[k := K_2])]\!]. \end{array}$$

4. $K_1 = ((\lambda k.K_3)\ K_4)$, then $\mathcal{K}^{-1}[\![K_2]\!][\mathcal{K}^{-1}[\![K_3[k := K_4]]\!]]$

$$\begin{array}{ll} \longrightarrow\!\!\!\!\!\rightarrow & \mathcal{K}^{-1}[\![K_3[k := K_4][k := K_2]]\!] \\ \equiv & \mathcal{K}^{-1}[\![K_3[k := K_4[k := K_2]]]\!] \\ \equiv & \mathcal{K}^{-1}[\![((\lambda k.K_3)\ K_4)[k := K_2]]\!]. \end{array}$$

5. $K_1 = \lambda x.P$: then, $\mathcal{K}^{-1}[\![K_2]\!][((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\ ])]$

$$\begin{array}{ll} \longrightarrow & ((\lambda x.\mathcal{K}^{-1}[\![K_2]\!][\mathcal{C}^{-1}[\![P]\!]])\ [\ ])(\beta_{lift}) \\ \twoheadrightarrow & ((\lambda x.\mathcal{C}^{-1}[\![P[k := K_2]]\!])\ [\ ]) \\ \equiv & \mathcal{K}^{-1}[\![(\lambda x.P)[k := K_2]]\!]. \end{array}$$

∎

Proof of case 1 in Lemma 14 (Page 25). Let $M \in \Lambda$, and $V \in \mathit{Values}$:

$$\boldsymbol{\lambda}\beta \vdash \mathcal{C}_k[\![((\lambda x.M)\ V)]\!] \twoheadrightarrow \mathcal{C}_k[\![M[x := V]]\!].$$

Proof: The main proof uses the following result that we state without proof.

$$\begin{array}{rcl} \boldsymbol{\lambda}\beta \vdash \mathcal{C}_k[\![M]\!][k := \mathcal{K}_k[\![E]\!]] & \twoheadrightarrow & \mathcal{C}_k[\![E[M]]\!] \\ \boldsymbol{\lambda}\beta \vdash \mathcal{K}_k[\![E_1]\!][k := \mathcal{K}_k[\![E]\!]] & \twoheadrightarrow & \mathcal{K}_k[\![E[E_1]]\!] \end{array}$$

For the main proof, we have by definition of $\mathcal{C}_k$,

$$\mathcal{C}_k[\![((\lambda x.M)\ V)]\!] = ((\lambda x.\mathcal{C}_k[\![M]\!])\ \Phi[\![V]\!]).$$

The latter term reduces to $\mathcal{C}_k[\![M]\!][x := \Phi[\![V]\!]]$. It remains to establish that substitution commutes with $\mathcal{C}_k$, *i.e.*,

1. $\boldsymbol{\lambda}\beta \vdash \mathcal{C}_k[\![M]\!][x := \Phi[\![V]\!]] \twoheadrightarrow \mathcal{C}_k[\![M[x := V]]\!]$.
2. $\boldsymbol{\lambda}\beta \vdash \mathcal{K}_k[\![E]\!][x := \Phi[\![V]\!]] \twoheadrightarrow \mathcal{K}_k[\![E[x := V]]\!]$.

The proof is by induction on the size of the argument to $\mathcal{C}_k$ or $\mathcal{K}_k$. Except for one case, the inductive hypothesis applies immediately. The interesting case occurs when $M = E[(x\ U)]$. The left hand side $\mathcal{C}_k[\![E[(x\ U)]]\!][x := \Phi[\![V]\!]]$

$$\begin{array}{ll} \equiv & ((x\ \mathcal{K}_k[\![E]\!])\ \Phi[\![U]\!])[x := \Phi[\![V]\!]] \\ \longrightarrow & ((\Phi[\![V]\!]\ \mathcal{K}_k[\![E]\!][x := \Phi[\![V]\!]])\ \Phi[\![U]\!][x := \Phi[\![V]\!]]) \\ \twoheadrightarrow & ((\Phi[\![V]\!]\ \mathcal{K}_k[\![E[x := V]]\!])\ \Phi[\![U[x := V]]\!]). \end{array}$$

The last line follows by cases on E if U is a variable. Otherwise, it follows by the inductive hypothesis. For readability, let $E' = E[x := V]$ and $U' = U[x := V]$. The goal is to prove that:

$$((\Phi[\![V]\!]\ \mathcal{K}_k[\![E']\!])\ \Phi[\![U']\!]) \twoheadrightarrow \mathcal{C}_k[\![E'[(V\ U')]]\!].$$

We proceed by cases of V:

1. $V = z$, then both sides are identical.

2. $V = \lambda z.L$. Then, $(((\lambda k.\lambda z.\mathcal{C}_k[\![L]\!])\ \mathcal{K}_k[\![E']\!])\ \Phi[\![U']\!])$

$$\begin{array}{lll} \longrightarrow & ((\lambda z.\mathcal{C}_k[\![L]\!][k := \mathcal{K}_k[\![E']\!]])\ \Phi[\![U']\!]) & (\beta) \\ \longrightarrow\!\!\!\!\!\rightarrow & ((\lambda z.\mathcal{C}_k[\![E'[L]]\!])\ \Phi[\![U']\!]) & \text{(aux. claim)} \\ \equiv & \mathcal{C}_k[\![E'[((\lambda z.L)\ U')]]\!]. & \end{array}$$

∎

Proof of Lemma 17 (Page 31). Let $P \in cps(\Lambda + callcc + \mathcal{A})$ and K be a continuation, and W be a value in the same language. Also, let $k_1, \ldots, k_n$ be the free continuation variables in these terms, and $k \notin \{k_1, \ldots, k_n\}$. Then,

1. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = P$.
2. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = K$.
3. $\boldsymbol{\lambda}\beta\eta \vdash (\Phi \circ \Phi^{-1})[\![W]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = W$.

Proof: The proof is by lexicographic induction on the number of abstractions of the form $\lambda k.K$ and the size of the terms. We proceed by cases:

1. $P = x$, then

$$\begin{array}{lll} (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] & \equiv & \mathcal{C}_k[\![(\mathcal{A}\ x)]\!] \\ & \equiv & (((\lambda k.\lambda x.x)\ k)\ x) \\ & \longrightarrow\!\!\!\!\!\rightarrow & x \qquad (\beta \text{ twice}) \end{array}$$

2. $P = \lambda k.K$, then

$$\begin{array}{lll} (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] & \equiv & \mathcal{C}_k[\![(\mathcal{A}\ \Phi^{-1}[\![\lambda k.K]\!])]\!] \\ & = & (((\lambda k.\lambda x.x)\ k)\ (\Phi \circ \Phi^{-1})[\![\lambda k.K]\!]) \\ & \longrightarrow\!\!\!\!\!\rightarrow & (\Phi \circ \Phi^{-1})[\![\lambda k.K]\!] \end{array}$$

The result follows by inlining the last case in the proof.

3. $P = (K\ W)$, then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![(K\ W)]\!] \equiv \mathcal{C}_k[\![\mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]]]\!]$ and by cases on K:
   (a) $K = k'$, then $\mathcal{C}_k[\![(k'\ \Phi^{-1}[\![W]\!])]\!] \equiv ((k'\ k)\ (\Phi \circ \Phi^{-1})[\![W]\!])$. By substituting every free continuation variable (in particular substituting k' by $(\lambda d.k')$) and using the inductive hypothesis, we get $(k'\ W)$.
   (b) $K = \lambda x.P'$, then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$

$$\begin{array}{ll} = & \mathcal{C}_k[\![((\lambda x.\mathcal{C}^{-1}[\![P']\!])\ \Phi^{-1}[\![W]\!])]\!] \\ = & ((\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P']\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]) \end{array}$$

The result follows by the inductive hypothesis.

(c) $K = W'K'$, then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$

$$\begin{array}{ll} \equiv & (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![((W'K')\ W)]\!] \\ \equiv & (((\Phi \circ \Phi^{-1})[\![W']\!]\ (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K']\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]) \end{array}$$

The result follows by the inductive hypothesis.

4. $K = k', \lambda x.P$ or WK. The cases are similar to the preceding three cases.
5. $W = x$, then $(\Phi \circ \Phi^{-1})[\![x]\!] \equiv x$.
6. $W = \lambda k.K$, then $(\Phi \circ \Phi^{-1})[\![\lambda k.K]\!]$

$$\begin{array}{ll} = & \Phi[\![\lambda z.\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K]\!][z]]\!] \\ = & \lambda k'.\lambda z.(((\lambda k''.\lambda u.(u\ k'')\ (\lambda d.k''))\ k')\ \Phi[\![\lambda k.\mathcal{K}^{-1}[\![K]\!][z]]\!]) \\ = & \lambda k'.\lambda z.((\lambda u.(u\ k')\ (\lambda d.k'))\ \Phi[\![\lambda k.\mathcal{K}^{-1}[\![K]\!][z]]\!]) \\ = & \lambda k'.\lambda z.((\lambda u.(u\ k')\ (\lambda d.k'))\ (\lambda k''.\lambda k.\mathcal{C}_{k''}[\![\mathcal{C}^{-1}[\![(K\ z)]\!]]\!])) \\ = & \lambda k'.\lambda z.\mathcal{C}_{k''}[\![\mathcal{C}^{-1}[\![(K\ z)]\!]]\!][k'' := k'][k := \lambda d.k'] \\ = & \lambda k'.\lambda z.\mathcal{C}_{k'}[\![\mathcal{C}^{-1}[\![(K\ z)]\!]]\!][k := \lambda d.k'] \\ = & \lambda k.\lambda z.\mathcal{C}_k[\![\mathcal{C}^{-1}[\![(K\ z)]\!]]\!][k := \lambda d.k] \end{array}$$

By substituting the remaining free continuation variables and using the inductive hypothesis (the term $(K\ z)$ has one less abstraction than $\lambda k.K$), we get $\lambda k.\lambda z.Kz$ which is equal to $\lambda k.K$.

∎

Proof of Lemma 18 (Page 31). Let $M \in \Lambda + \mathit{callcc} + \mathcal{A}$, and E be an evaluation context in the same language, and k a variable that is not free in either. Then,

- $\boldsymbol{\lambda}\beta_v XC \vdash (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!] = (k\ M)$
- $\boldsymbol{\lambda}\beta_v XC \vdash (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!] = (k\ E)$

Proof: By induction on the size of the terms. We proceed by cases:

1. $M = V$, then $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![V]\!] \equiv \mathcal{C}^{-1}[\![(k\ \Phi[\![V]\!])]\!] \equiv (k\ (\Phi^{-1} \circ \Phi)[\![V]\!])$. By cases, we show that $(\Phi^{-1} \circ \Phi)[\![V]\!] = V$.

   (a) $V = x$, then $(\Phi^{-1} \circ \Phi)[\![x]\!] \equiv x$.

   (b) $V = \lambda x.N$, then

$$\begin{array}{rll} (\Phi^{-1} \circ \Phi)[\![V]\!] & \equiv & \Phi^{-1}[\![\lambda k.\lambda x.\mathcal{C}_k[\![N]\!]]\!] \\ & \equiv & \lambda z.\mathit{callcc}\ \lambda k.((\lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![N]\!])\ z) \\ & = & \lambda z.\mathit{callcc}\ \lambda k.((\lambda x.(k\ N))\ z) \\ & & \quad \text{(induction)} \\ & = & \lambda z.\mathit{callcc}\ \lambda k.k\ ((\lambda x.N)\ z) \qquad (\beta_{lift}) \\ & = & \lambda z.((\lambda x.N)\ z) \\ & & \quad (C_{current} \text{ and } C_{elim}) \\ & = & \lambda x.N \qquad (\beta_v) \end{array}$$

(c) $V = callcc$, then $(\Phi^{-1} \circ \Phi)[\![V]\!]$

$$
\begin{array}{lll}
\equiv & \Phi^{-1}[\![\lambda k.\lambda u.(u\ k)\ \lambda d.k]\!] & \\
\equiv & \lambda z.callcc\ \lambda k.((\lambda u.(k\ (u\ \lambda f.callcc\ \lambda d.(k\ f))))\ z) & \\
= & \lambda z.callcc\ \lambda k.((\lambda u.(k\ (u\ \lambda f.(k\ f))))\ z) & (C_{elim}) \\
= & \lambda z.callcc\ \lambda k.((\lambda u.(k\ (u\ k)))\ z) & (\eta_v) \\
= & \lambda z.callcc\ \lambda k.(k\ (z\ k)) & (\beta_v) \\
= & \lambda z.callcc\ \lambda k.(z\ k) & (C_{current}) \\
= & callcc & (\eta_v \text{ twice})
\end{array}
$$

(d) $V = \mathcal{A}$, then

$$
\begin{array}{llll}
(\Phi^{-1} \circ \Phi)[\![V]\!] & \equiv & \Phi^{-1}[\![\lambda k.\lambda x.x]\!] & \\
& = & \lambda z.callcc\ \lambda k.((\lambda x.\mathcal{A}\ x)\ z) & \\
& = & \lambda z.((\lambda x.\mathcal{A}\ x)\ z) & (C_{elim}) \\
& = & (\lambda z.\mathcal{A}\ z) & (\beta_v) \\
& = & \mathcal{A} & (\eta_v)
\end{array}
$$

2. $M = E[(V_1\ V_2)]$, then $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$

$$
\begin{array}{ll}
= & \mathcal{C}^{-1}[\![((\Phi[\![V_1]\!]\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V_2]\!])]\!] \\
= & \mathcal{K}^{-1}[\![(\Phi[\![V_1]\!]\ \mathcal{K}_k[\![E]\!])]\!][(\Phi^{-1} \circ \Phi)[\![V_2]\!]] \\
= & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!][((\Phi^{-1} \circ \Phi)[\![V_1]\!]\ (\Phi^{-1} \circ \Phi)[\![V_2]\!])]
\end{array}
$$

The result follows by the inductive hypothesis and a repetition of the argument in case 1.

3. $E = [\,]$, then $(\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!] \equiv \mathcal{K}^{-1}[\![k]\!] \equiv (k\ [\,])$.

4. $E = E_1[(V\ [\,])]$, then $(\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!]$

$$
\begin{array}{ll}
= & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[(V\ [\,])]]\!] \\
= & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1]\!][((\Phi^{-1} \circ \Phi)[\![V]\!]\ [\,])]
\end{array}
$$

The result follows by the inductive hypothesis and a repetition of the argument in case 1.

5. $E = E_1[([\,]\ M)]$, then

$$
\begin{array}{llll}
(\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!] & \equiv & (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[([\,]\ M)]]\!] & \\
& \equiv & ((\lambda f.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![E_1[(f\ M)]]\!])\ [\,]) & \\
& = & ((\lambda f.(k\ E_1[(f\ M)]))\ [\,]) & (\text{induction}) \\
& = & (k\ E_1[([\,]\ M)]) & (\beta'_\Omega)
\end{array}
$$

∎

Proof of case 1 in Lemma 19 (Page 32). Let $k_1 \ldots k_n$ be the free continuation variables in P and W. Then,

$$
\begin{array}{l}
\boldsymbol{\lambda}\beta_v XC \vdash callcc\ \lambda k_1. \ldots callcc\ \lambda k_n.\mathcal{C}^{-1}[\![((\lambda x.P)\ W)]\!] = \\
\qquad callcc\ \lambda k_1. \ldots callcc\ \lambda k_n.\mathcal{C}^{-1}[\![P[x := W]]\!]
\end{array}
$$

Proof: Applying $\mathcal{C}^{-1}$ to $((\lambda x.P)\ W)$ yields:

$$\mathcal{K}^{-1}[\![(\lambda x.P)]\!][\Phi^{-1}[\![W]\!]] \equiv ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ \Phi^{-1}[\![W]\!]).$$

By β_v, we get $\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]]$. It remains to show that $\mathcal{C}^{-1}$ commutes with the substitution. We prove the following statements by induction on the structure of the terms:

- $\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]] \equiv \mathcal{C}^{-1}[\![P[x := W]]\!]$.
- $\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]] \equiv \Phi^{-1}[\![W'[x := W]]\!]$.
- $\mathcal{K}^{-1}[\![K]\!][x := \Phi^{-1}[\![W]\!]] \equiv \mathcal{K}^{-1}[\![K[x := W]]\!]$.

1. $P = y$, then $\mathcal{C}^{-1}[\![y]\!][x := \Phi^{-1}[\![W]\!]] \equiv (\mathcal{A}\ y) \equiv \mathcal{C}^{-1}[\![y[x := W]]\!]$.
2. $P = x$, then
$$\mathcal{C}^{-1}[\![x]\!][x := \Phi^{-1}[\![W]\!]] \equiv (\mathcal{A}\ \Phi^{-1}[\![W]\!]) \equiv \mathcal{C}^{-1}[\![x[x := W]]\!]$$
3. $P = \lambda k.K$, then $\mathcal{C}^{-1}[\![\lambda k.K]\!][x := \Phi^{-1}[\![W]\!]]$
$$\begin{array}{ll} \equiv & (\mathcal{A}\ \Phi^{-1}[\![\lambda k.K]\!][x := \Phi^{-1}[\![W]\!]]) \\ \equiv & (\mathcal{A}\ \Phi^{-1}[\![\lambda k.K[x := W]]\!]) \qquad \text{(similar to case 7)} \\ \equiv & \mathcal{C}^{-1}[\![\lambda k.K[x := W]]\!] \end{array}$$
4. $P = (K'\ W')$, then $\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]]$
$$\begin{array}{ll} \equiv & \mathcal{K}^{-1}[\![K']\!][x := \Phi^{-1}[\![W]\!]][\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]]] \\ \equiv & \mathcal{K}^{-1}[\![K'[x := W]]\!][\Phi^{-1}[\![W'[x := W]]\!]] \qquad \text{(induction)} \\ \equiv & \mathcal{C}^{-1}[\![(K'\ W')[x := W]]\!] \end{array}$$
5. $W' = x$, then
$$\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]] \equiv \Phi^{-1}[\![W]\!] \equiv \Phi^{-1}[\![x[x := W]]\!]$$
6. $W' = y$, then $\Phi^{-1}[\![y]\!][x := \Phi^{-1}[\![W]\!]] \equiv y \equiv \Phi^{-1}[\![y[x := W]]\!]$.
7. $W' = \lambda k.K$, then $\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]]$
$$\begin{array}{ll} \equiv & \lambda z.\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K]\!][x := \Phi^{-1}[\![W]\!]][z] \\ \equiv & \lambda z.\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K[x := W]]\!][z] \qquad \text{(induction)} \\ \equiv & \Phi^{-1}[\![\lambda k.K[x := W]]\!] \end{array}$$
8. $K = k$, then $\mathcal{K}^{-1}[\![k]\!][x := \Phi^{-1}[\![W]\!]] \equiv (k\ [\,]) \equiv \mathcal{K}^{-1}[\![k[x := W]]\!]$.
9. $K = \lambda y.P$, then
$$\mathcal{K}^{-1}[\![\lambda y.P]\!][x := \Phi^{-1}[\![W]\!]] \equiv ((\lambda y.\mathcal{C}^{-1}[\![P]\!][x := \Phi^{-1}[\![W]\!]])\ [\,])$$

The result follows by induction.

10. $K = W'K'$, then the result follows also by a straightforward application of the inductive hypothesis.

∎

Proof of case 4 in Lemma 19 (Page 32). Let $k_1 \ldots k_n$ be the free continuation variables in $((\lambda k.K_1)\ K_2)$. Then,

$$\boldsymbol{\lambda}\beta_v XC \vdash \mathit{callcc}\ \lambda k_1. \ldots \mathit{callcc}\ \lambda k_n.\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!] = \mathit{callcc}\ \lambda k_1. \ldots \mathit{callcc}\ \lambda k_n.\mathcal{K}^{-1}[\![K_1[k := K_2]]\!]$$

Proof: Let D be a context $\mathit{callcc}\ \lambda k_1. \ldots \mathit{callcc}\ \lambda k_n.C[\ ]$ where C is an arbitrary context, then it suffices to prove:

$$\boldsymbol{\lambda}\beta_v XC \vdash D[\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!]] = D[\mathcal{K}^{-1}[\![K_1[k := K_2]]\!]]$$

The left hand side $D[\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!]]$

$$\begin{array}{cl} \equiv & D[\mathcal{K}^{-1}[\![K_2]\!][(\Phi^{-1}[\![\lambda k.K_1]\!]\ [\ ])]] \\ = & D[\mathcal{K}^{-1}[\![K_2]\!][((\lambda z.\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K_1]\!][z])\ [\ ])]] \end{array}$$

It remains to prove the following statements by induction on the structure of P, W, and K_1.

$$\begin{array}{l} \boldsymbol{\lambda}\beta_v XC \vdash D[\mathcal{C}^{-1}[\![P[k := K_2]]\!]] = D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{C}^{-1}[\![P]\!]]] \\ \boldsymbol{\lambda}\beta_v XC \vdash D[\Phi^{-1}[\![W[k := K_2]]\!]] = D[\Phi^{-1}[\![W]\!][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]] \\ \boldsymbol{\lambda}\beta_v XC \vdash D[\mathcal{K}^{-1}[\![K_1[k := K_2]]\!]] = \\ \qquad\qquad D[\mathcal{K}^{-1}[\![K_2]\!][((\lambda z.\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K_1]\!][z])\ [\ ])]] \end{array}$$

The main proof relies on the following auxiliary claim that we state without proof.

Let K be a continuation in $cps(\Lambda + \mathit{callcc} + \mathcal{A})$ with free continuation variables $k_1, \ldots, k_n$, then either:

$$\begin{array}{rcll} \boldsymbol{\lambda}\beta_v XC \vdash \mathcal{K}^{-1}[\![K]\!] & = & (\mathcal{A}\ E) & \\ \text{or } \boldsymbol{\lambda}\beta_v XC \vdash \mathcal{K}^{-1}[\![K]\!] & = & (k_i\ E) & 1 \leq i \leq n \end{array}$$

The auxiliary claim implies that for any evaluation context E:

$$D[E[\mathcal{K}^{-1}[\![K]\!][M]]] = D[\mathcal{K}^{-1}[\![K]\!][M]].$$

The latter result implies that $D[\mathcal{K}^{-1}[\![K]\!][\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K']\!][M]]]$

$$\begin{array}{cl} = & D[\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K]\!][((\lambda k.\mathcal{K}^{-1}[\![K']\!][M])\ \lambda f.(k'\ \mathcal{K}^{-1}[\![K]\!][f]))]] \\ = & D[\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K]\!][((\lambda k.\mathcal{K}^{-1}[\![K']\!][M])\ \lambda f.\mathcal{K}^{-1}[\![K]\!][f])]] \\ = & D[\mathcal{K}^{-1}[\![K]\!][((\lambda k.\mathcal{K}^{-1}[\![K']\!][M])\ \lambda f.\mathcal{K}^{-1}[\![K]\!][f])]] \\ = & D[\mathcal{K}^{-1}[\![K]\!][\mathcal{K}^{-1}[\![K']\!][((\lambda k.M)\ \lambda f.\mathcal{K}^{-1}[\![K]\!][f])]]] \\ = & D[\mathcal{K}^{-1}[\![K']\!][((\lambda k.M)\ \lambda f.\mathcal{K}^{-1}[\![K]\!][f])]] \\ = & D[((\lambda k.\mathcal{K}^{-1}[\![K']\!][M])\ \lambda f.\mathcal{K}^{-1}[\![K]\!][f])] \end{array}$$

The main proof proceeds by case analysis:

1. $P = x$, then:

$$\begin{aligned} D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{A}\ x]] &= D[\mathcal{K}^{-1}[\![K_2]\!][\mathcal{A}\ x]] \\ &= D[(\mathcal{A}\ x)] \\ &= D[\mathcal{C}^{-1}[\![x]\!]] \end{aligned}$$

2. $P = \lambda k'.K'$. The left hand side:

$$\begin{aligned} & D[\mathcal{C}^{-1}[\![\lambda k'.K'[k := K_2]]\!]] \\ = \ & D[(\mathcal{A}\ \Phi^{-1}[\![\lambda k'.K'[k := K_2]]\!])] \\ = \ & D[(\mathcal{A}\ \lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K']\!][z]])] \\ = \ & D[(\mathcal{A}\ \lambda z.\mathit{callcc}\ \lambda k'.((\lambda k.\mathcal{K}^{-1}[\![K']\!][z])\ \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]))] \\ & \quad \text{(auxiliary claim)} \\ = \ & D[(\mathcal{A}\ \lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K']\!][z])[k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]] \\ = \ & D[((\lambda k.(\mathcal{A}\ \lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K']\!][z]))\ (\lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]))] \\ = \ & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.(\mathcal{A}\ \lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K']\!][z])]] \\ & \quad \text{(auxiliary claim)} \\ = \ & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{C}^{-1}[\![\lambda k'.K']\!]]] \end{aligned}$$

3. $P = (K\ W)$. The left hand side $D[\mathcal{C}^{-1}[\![(K\ W)[k := K_2]]\!]]$

$$\begin{aligned} = \ & D[\mathcal{K}^{-1}[\![K[k := K_2]]\!][\Phi^{-1}[\![W[k := K_2]]\!]]] \\ = \ & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k. \\ & \qquad \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]]]] \\ = \ & \mathrm{D}[((\lambda k.\mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]]) \\ & \quad (\lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]))] \\ = \ & D[\mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]] \\ = \ & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]]]] \end{aligned}$$

4. $W = x$, then the result is immediate.
5. $W = \lambda k'.K$, then $\Phi^{-1}[\![\lambda k'.K[k := K_2]]\!]$

$$\begin{aligned} = \ & D[\lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K[k := K_2]]\!][z]] \\ = \ & D[\lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K]\!][z]]] \\ = \ & D[\lambda z.\mathit{callcc}\ \lambda k'.\mathcal{K}^{-1}[\![K]\!][z][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]] \end{aligned}$$

6. $K_1 = k'$, then:

$$\begin{aligned} D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.(k'\ [\,])]] &= D[\mathcal{K}^{-1}[\![K_2]\!][(k'\ [\,])]] \\ &= D[(k'\ [\,])] \\ &= D[\mathcal{K}^{-1}[\![k']\!]] \end{aligned}$$

7. $K_1 = k$, then $D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.(k\ [\,])]] = D[\mathcal{K}^{-1}[\![K_2]\!]]$.
8. $K_1 = \lambda x.P$, then $D[\mathcal{K}^{-1}[\![\lambda x.P[k := K_2]]\!]]$

$$\begin{aligned} = \ & D[((\lambda x.\mathcal{C}^{-1}[\![P[k := K_2]]\!])\ [\,])] \\ = \ & D[((\lambda x.\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{C}^{-1}[\![P]\!]])\ [\,])] \\ = \ & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\,])]] \end{aligned}$$

9. $K_1 = W'K'$, then the left hand side:

$$
\begin{array}{ll}
& D[\mathcal{K}^{-1}[\![W'K'[k := K_2]]\!]] \\
= & D[\mathcal{K}^{-1}[\![K'[k := K_2]]\!][(\Phi^{-1}[\![W'[k := K_2]]\!]\ [\ ])]] \\
= & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k. \\
& \qquad\qquad \mathcal{K}^{-1}[\![K']\!][(\Phi^{-1}[\![W']\!][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]] \\
& \qquad\qquad\qquad [\ ])]]] \\
= & D[\mathcal{K}^{-1}[\![K']\!][(\Phi^{-1}[\![W']\!]\ [\ ])][k := \lambda f.\mathcal{K}^{-1}[\![K_2]\!][f]]] \\
= & D[\mathcal{K}^{-1}[\![K_2]\!][\mathit{callcc}\ \lambda k.\mathcal{K}^{-1}[\![K']\!][(\Phi^{-1}[\![W']\!]\ [\ ])]]]
\end{array}
$$

∎

References

1. Allison, L. *A Practical Introduction to Denotational Semantics.* Volume 23 of *Cambridge Computer Science Texts*, Cambridge University Press (1986).

2. Appel, A. *Compiling with Continuations.* Cambridge University Press (1992).

3. Appel, A. and Jim, T. Continuation-passing, closure-passing style. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages* (1989) 293–302.

4. Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, revised edition (1984).

5. Bartley, D. H. and Jensen, J. C. The implementation of PC Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1986) 86–93.

6. Boehm, H.-J. and Demers, A. Implementing Russel. In *Proceedings of the ACM Sigplan Symposium on Compiler Construction*, Sigplan Notices, 21, 7 (1986) 186–195.

7. Bondorf, A. Improving binding times without explicit CPS-conversion. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1992) 1–10.

8. Clinger, W., Friedman, D., and Wand, M. A scheme for a higher-level semantic algebra. In Reynolds, J. and Nivat, M., editors, *Algebraic Methods in Semantics*, Cambridge University Press (1985) 237–250.

9. Crank, E. and Felleisen, M. Parameter-passing and the lambda calculus. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages* (1991) 233–244.

10. Curry, H. B. and Feys, R. *Combinatory Logic, Volume I.* North-Holland, Amsterdam (1958).

11. Danvy, O. Back to direct style. *Science of Computer Programming* (1993). To appear. Preliminary version in: Proceedings of the 4th European Symposium on Programming, 1992. Lecture Notes in Computer Science, 582, Springer Verlag.

12. Danvy, O. and Filinski, A. Abstracting control. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1990) 151–160.

13. Danvy, O. and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2, 4 (1992) 361–391.

14. Danvy, O. and Lawall, J. Back to direct style II: First-class continuations. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1992) 299–310.

15. Felleisen, M. The theory and practice of first-class prompts. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages* (1988) 180–190.

16. Felleisen, M. and Friedman, D. P. Control operators, the SECD-machine, and the λ-calculus. In Wirsing, M., editor, *Formal Description of Programming Concepts-III*, North-Holland (1986) 193–217.

17. Felleisen, M. and Friedman, D. P. A syntactic theory of sequential state. *Theoretical Computer Science*, 69, 3 (1989) 243–287. Preliminary version in: Conference Record of the 14th ACM Symposium on Principles of Programming Languages, 1987.

18. Felleisen, M. and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102 (1992) 235–271. Technical Report 89-100, Rice University.

19. Felleisen, M., Friedman, D. P., Kohlbecker, E., and Duba, B. A syntactic theory of sequential control. *Theoretical Computer Science*, 52, 3 (1987) 205–237. Preliminary version: Reasoning with Continuations, in Proceedings of the 1st IEEE Symposium on Logic in Computer Science, 1986.

20. Felleisen, M., Wand, M., Friedman, D.P., and Duba, B.F. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1988) 52–62.

21. Fischer, M. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs*, Sigplan Notices, 7, 1 (1972) 104–109.

22. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1993). To appear.

23. Friedman, D. P., Haynes, C.T., and Kohlbecker, E. Programming with continuations. In Pepper, P., editor, *Program Transformations and Programming Environments*, Springer-Verlag (1985) 263–274.

24. Friedman, D. P., Wand, M., and Haynes, C.T. *Essentials of Programming Languages.* The MIT Press (1992).

25. Galbiati, L. and Talcott, C. L. A simplifier for untyped lambda expressions. In *Proceedings Conditional Term Rewriting Systems*, Lecture Notes in Computer Science, 516 (1990) 342–353.

26. Griffin, T.G. A formulae-as-types notion of control. In *Conference Record of the 17th ACM Symposium on Principles of Programming Languages* (1990) 47–58.

27. Haynes, C.T., Friedman, D. P., and Wand, M. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press), 11, 3/4 (1986) 143–153.

28. Hieb, R., Dybvig, K., and Bruggeman, C. Representing control in the presence of first-class continuations. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1990) 66–77.

29. Kelsey, R. and Hudak, P. Realistic compilation by program transformation. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages* (1989) 281–292.

30. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM Sigplan Symposium on Compiler Construction*, Sigplan Notices, 21, 7 (1986) 219–233.

31. Landin, P.J. The mechanical evaluation of expressions. *Computer Journal*, 6, 4 (1964) 308–320.

32. Leroy, X. *The Zinc Experiment: An Economical Implementation of the ML Language*. Technical Report 117, INRIA (1990).

33. Mason, I. and Talcott, C. L. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1, 3 (July 1991) 287–327.

34. Mazurkiewicz, A.W. Proving algorithms by tail functions. *Information and Control*, 18 (1971) 220–226.

35. Meyer, A. R. and Wand, M. Continuation semantics in typed lambda-calculi. In *Proceedings Workshop Logics of Programs*, Lecture Notes in Computer Science, 193 (1985) 219–224.

36. Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science* (1989) 14–23. Also appeared as: LFCS Report ECS-LFCS-88-86, University of Edinburgh, 1988.

37. Morris, L. The next 700 formal language descriptions. *Lisp and Functional Programming* (1993). To appear.

38. Nielson, F. A denotational framework for data flow analysis. *Acta Informatica*, 18 (1982) 265–287.

39. Plotkin, G.D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1 (1975) 125–159.

40. Rees, J. and Clinger, W. Revised^3 report on the algorithmic language Scheme. *Sigplan Notices*, 21, 12 (1986) 37–79.

41. Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference* (1972) 717–740.

42. Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1992) 288–298. Technical Report 92-180, Rice University.

43. Shivers, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University (1991).

44. Sitaram, D. and Felleisen, M. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1990) 161–175.

45. Steel, T.B., editor. *Formal Language Description Languages for Computer Programming.* North-Holland (1966).

46. Steele, G. L. *Rabbit: A Compiler for Scheme.* MIT AI Memo 474, Massachusetts Institute of Technology (1978).

47. Strachey, C. and Wadsworth, C.P. *Continuations: A Mathematical Semantics for Handling Full Jumps.* Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group (1974).

48. Talcott, C. L. A theory for program and data specification. *Theoretical Computer Science*, 104 (1992) 129–159. Preliminary version in Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, (Lecture Notes in Computer Science, 429, 1990).

49. Wand, M. Correctness of procedure representations in higher-order assembly language. In Brookes, S., editor, *Proceedings of the Conference on the Mathematical Foundations of Programing Semantics*, Lecture Notes in Computer Science, 598, Springer Verlag (1992) 294–311.