RICE UNIVERSITY

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

by

Amr A. Sabry

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

Doctor of Philosophy

Approved, Thesis Committee:

Matthias Felleisen, Chairman
Professor of Computer Science

Robert Cartwright
Professor of Computer Science

Keith Cooper
Associate Professor of Computer Science

Steve Cox
Assistant Professor of Computational and Applied Mathematics

Houston, Texas

August, 1994

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

Amr A. Sabry

Abstract

Compilers for higher-order programming languages like Scheme, ML, and Lisp can be broadly characterized as either "direct compilers" or "continuation-passing style (CPS) compilers", depending on their main intermediate representation. Our central result is a precise correspondence between the two compilation strategies.

Starting from the theoretical foundations of direct and CPS compilers, we develop relationships between the main components of each compilation strategy: generation of the intermediate representation, simplification of the intermediate representation, code generation, and data flow analysis. For each component, our results pinpoint the superior compilation strategy, the reason for which it dominates the other strategy, and ways to improve the inferior strategy. Furthermore, our work suggests a synthesis of the direct and CPS compilation strategies that combines the best aspects of each.

The contributions of this thesis include a comprehensive analysis of the properties of the CPS intermediate representation, a new optimal CPS transformation and its inverse, a new intermediate representation for direct compilers, an equivalence between the canonical equational theories for reasoning about continuations and general computational effects, a sound and complete equational axiomatization of the semantics of call-by-value control operators, a methodology for deriving equational logics for imperative languages, and formal relationships between code generators and data flow analyzers for direct and CPS compilers. These contributions unify concepts in two distinct compilation strategies, and can be used to compare specific compilers.

Acknowledgments

I would like to thank my friend and advisor Matthias Felleisen for his guidance. Matthias provided a challenging and exciting atmosphere for both teaching and research; his ideas, technical expertise, and meticulous comments were invaluable throughout the years.

My interaction with the rest of my committee members was extremely helpful. Without any doubt, Robert (Corky) Cartwright's experience and vision helped shape mine. Keith Cooper provided an alternative and unbiased point of view for my research. I am also grateful to Steve Cox for his comments and contributions.

In general, the programming language group at Rice University provided a healthy and stimulating environment. I would like to extend my thanks to all the members especially Bruce Duba, Cormac Flanagan, Andrew Wright, and John Gateley. Bruce's practical expertise and cooperation were helpful at various stages of my research. Cormac's keen criticism made me re-evaluate a number of my assertions; his effort in the implementation in Chapter 5 is greatly appreciated. As for Andrew, it suffices to say that we came to Rice on the same day, have shared an office since then, and are scheduled to leave at the same time. My relationship to John Gateley is even closer; one day, he invited his Egyptian friend to spend Christmas with his family. A year later, he became my brother-in-law!

During my last academic year, Matthias decided to spend his sabbatical at Carnegie Mellon University; Peter Lee and Bob Harper kindly invited me to come along for what turned out to be a most profitable and exciting year. My appreciations to all the people at CMU specially Nevin Heintze for the interesting discussions and meetings.

My kind thanks are also directed to Olivier Danvy, Carolyn Talcott, and John Field for numerous technical discussions and advice.

More personally, I would like to acknowledge the various contributions of my friends and family. My long time friend, Mootaz El-Nozahy, encouraged me to join him at Rice University, and could tolerate being around me more than anybody else. Amr Abouhashish and Maged Hamed came close in our all-night bridge parties.

My parents, my brother, and my parents in-laws all helped me in various important ways; my love to all of them. Most importantly, my partner and wife Elizabeth supported me beyond what my poor words can express.

Finally, this research would not have been possible without the generous support of Rice University, Carnegie Mellon University, the state of Texas, the Professional Activities Committee of the Association for Computing Machinery's Special Interest Group on Programming Languages, and the National Science Foundation.

Contents

Illustrations

Chapter 1

Background and Motivation

The continuation-passing style (CPS) transformation is ubiquitous in the area of programming language semantics:

- it defines the intermediate representation of many compilers and program analysis tools [3, 15, 39, 55, 56, 58, 82, 92, 97];
- it is essential in the construction of many denotational models of programming languages [99, 101];
- it permits a simulation of call-by-value semantics using call-by-name semantics [77];
- it converts heap allocation into stack allocation [5, 37];
- it embeds classical logic into constructive logic [46, 70, 71].

In each of these application domains, the relationship between the original programs and their CPS transforms specializes to a problem of independent interest, *e.g.*, relating source programs to their intermediate representation in a CPS compiler, relating direct compilers to CPS compilers, relating classical proofs to constructive proofs, and relating denotations in direct semantics to denotations in continuation semantics [65, 78, 91, 98].

Some of the above problems have been thoroughly studied in the literature, but the precise role of the CPS transformation in the compilation process is still not completely understood. The reason for this gap in our knowledge may be due to the size, complexity, and informal nature of most compilers; it is certainly not due to a lack of interest in the subject. Indeed many state-of-the-art compilers for applicative higher-order programming languages (Scheme, ML, Lisp, etc) use a CPS intermediate representation [3, 56, 82], and many others do not [7, 9, 12, 50, 59]. Informal or incomplete arguments regarding the relative advantages and disadvantages of the

CPS intermediate representation with respect to other intermediate representations are also common:

- The CPS transformation translates complicated control facilities in the source language, *e.g.*, exception handlers and *call-with-current-continuation* [13] to simple procedures that manipulate their continuation arguments in non-standard ways [77, 101].
- The CPS intermediate representation is simple: it consists only of basic primitive operations and procedure applications, and has a well-understood semantics in terms of the λ-calculus [75].
- The canonical equational theory for the CPS intermediate representation (the λ-calculus) proves more equations, hence more optimizations, than the canonical equational theory for the call-by-value source language [75].
- The CPS transformation systematically introduces names for every intermediate computation and re-orders the subexpressions of a program in a way that reflects the order in which they should be evaluated [39].
- Despite its functional appearances, CPS code constitutes an (imperative) abstract assembly language whose standard reduction sequence mimics the behavior of typical target machines [3, 4, 41, 55, 104].
- Data-flow analyzers, partial evaluators, and other tools perform better on CPS programs than on source programs [22, 33, 34, 72].
- The CPS transformation is a global transformation that affects every subexpression in a program. It re-structures 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].
- The CPS transformation is a convenient organizational principle that simplifies the construction of compilers.

To illustrate some of the above points, we consider the following (naïve) example. The expression $(+\ 1\ (*\ (f\ 2)\ 3))$ can be read as "call the function f with argument

2, multiply the result by 3, and then add 1." Applying any of the conventional CPS transformations to the expression yields:

$$(f\ 2\ (\lambda v_1.(*_k\ v_1\ 3\ (\lambda v_2.(+_k\ 1\ v_2\ (\lambda v_3.v_3)))))),$$

where $(\lambda x.M)$ denotes a procedure whose parameter is x and body is M, and the procedures $*_k$ and $+_k$ denote the CPS counterparts of the usual multiplication and addition procedures respectively. Since it takes some amount of training to read CPS programs, we rewrite the expression using a more appealing layout:

$$\begin{array}{ll}(f\ 2 & (\lambda v_1. \\ (*_k\ v_1\ 3 & (\lambda v_2. \\ (+_k\ 1\ v_2 & (\lambda v_3. \\ v_3)))))). & \end{array}$$

The expression becomes even more readable when we replace the symbol λ with the notation for assignment $\rightarrow$:

$$\begin{array}{lcl}\mathit{call}\ f\ 2 & \rightarrow & v_1 \\ *\ v_1\ 3 & \rightarrow & v_2 \\ +\ 1\ v_2 & \rightarrow & v_3 \\ \mathit{return}\ v_3 & & \end{array}$$

This last expression is essentially the conventional triples representation for our expression [1].

The example hints that the CPS transformation generates a sensible intermediate representation. In addition, given that the CPS transformation is a simple, well-understood, linear transformation that uniformly applies to all programming constructs, it is not surprising that it became a popular tool in compilers.

This thesis is a formal analysis of the properties of the CPS compilation strategy and its connection to other more conventional compilation techniques. The precise results are discussed in the next section, which also serves as an outline of the thesis. Following the outline, Section 2 introduces the technical background and notation used throughout the thesis.

1.1 Outline of the Thesis

The thesis is composed of 7 chapters including this one.

Chapter 2 is a mostly historical account of the discovery and evolution of CPS transformations. It stresses that the "spirit" of the CPS transformation is to represent *all* transfers of control uniformly: *e.g.*, in a CPS program, statement sequencing, jumps, procedure calls, procedure returns, and coroutines are all encoded using the same mechanism. The chapter also includes a comprehensive study of the administrative reductions in CPS transformations and concludes with a new optimal CPS transformation derived from our analysis.

Chapter 3 addresses the most elementary and most obvious advantage of CPS compilers over direct compilers, namely that the equational theory for the CPS intermediate representation is richer than the equational theory for pure call-by-value languages [75]. Since equational theories are commonly used to perform simple optimizations [3], it appears that CPS compilers can perform more simplifications on their intermediate representation than direct compilers. In order to understand and report these optimizations of CPS programs in terms of transformations on the original source programs, we formulate a precise correspondence between the equational theories for both languages. In the process of proving the result, we establish a number of intermediate results that are of independent interest. The main novel contributions of this chapter include a new inverse CPS transformation, a correspondence between general computational effects and continuations, and the development of a calculus for call-by-value languages that is sound and complete with respect to equivalence in a certain class of denotational models.

Chapter 4 extends the results of Chapter 3 to realistic languages that include a variety of constructs including sophisticated control operators, control delimiters, assignments, conditionals, basic and functional constants. The main result is an equational theory for the core of a Scheme-like language that is as powerful as the equational theory for the corresponding CPS programs. The main novel contribution of this chapter is a sound and complete axiomatization of the semantics of call-by-value control operators.

Chapter 5 analyzes the correspondence between the code generation phases in direct and CPS compilers. The correspondence is based on a formal development of the abstract machines for both compilers. The analysis of CPS compilers shows that a typical compiler performs the following conceptual sequence of steps: first convert the source program to CPS using a naïve algorithm, then simplify the resulting intermediate term using the equational theory for CPS terms, and finally optimize the management of continuations during code generation. The original observation at this point is that the last step effectively performs an inverse CPS transformation similar to the one developed in Chapter 4. Given our previous results, we conclude that the three phases of a CPS compiler are equivalent to a simple source-to-source translation that can be performed using a fragment of our complete equational theory for call-by-value languages.

Chapter 6 addresses the correspondence between the data-flow analysis phases in direct and CPS compilers. The analysis is motivated by the common belief that the CPS transformation has a positive effect on the analysis of programs. To investigate these claims, we derive canonical data flow analyzers for both direct and CPS compilers and formally compare their outputs. Under the assumption that the analyses are "naturally" derived from the semantics of the languages, we establish that the results of a direct analysis are *incomparable* to the results of an analysis of the equivalent CPS program. In other words, the translation of the source program to a CPS version may increase or decrease static information. The gain of information occurs in non-distributive analyses and is solely due to the *duplication* of the analysis of the continuation. The loss of information is due to the confusion of distinct procedure returns.

Chapter 7 concludes this dissertation with a summary of contributions, open problems, and directions for future research.

1.2 Λ: Syntax, Calculi, and Semantics

The core of our source language is a typed or untyped call-by-value functional language based on the language Λ of the pure lambda calculus [6]. The latter language consists of variables, λ-abstractions (procedures) and applications. The set of terms M is generated inductively over an infinite set of variables *Vars*:

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

We adopt Barendregt's [6, ch 2,3] notation and terminology for this 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; the set of closed terms is denoted by Λ^0. 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 expression $M[x := N]$ is the result of the capture-free substitution of all free occurrences of x in M by N. For example, $(\lambda x.xz)[z := (\lambda y.x)] \equiv (\lambda u.u(\lambda y.x))$. A *context*, C, is a term with a "hole", [ ], in the place of one subexpression. The operation of *filling* the context C with an expression M yields the term $C[M]$, possibly capturing some free variables of M in the process. Thus, the result of filling $(\lambda x.x[\,])$ with $(\lambda y.x)$ is $(\lambda x.x(\lambda y.x))$.

1.2.1 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 are the following notions of reductions:

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

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

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

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 $\boldsymbol{\lambda}A$ to refer to the theory generated by a set of axioms A. When a theory $\boldsymbol{\lambda}A$ proves an equation $M = N$, we write $\boldsymbol{\lambda}A \vdash M = N$. If the proof does not involve the inference rule *Symmetry*, we write $\boldsymbol{\lambda}A \vdash M \twoheadrightarrow N$.

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., $\boldsymbol{\lambda}R \vdash M \twoheadrightarrow L$ and $\boldsymbol{\lambda}R \vdash N \twoheadrightarrow L$. A term M is in R-normal form if there are no R-reductions starting with M.

As described in the rest of this chapter, equational theories can be used to specify the semantics of programming languages as well as to reason about compiler optimizations such as in-lining and constant propagation.

1.2.2 Semantics

The semantics of the language Λ is a 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. A common method for specifying the semantics of Λ is based on the Curry-Feys standard reduction theorem [27, 75]. 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 context E and a leftmost-outermost redex R (not inside an abstraction), and (ii) fills E with the contractum of R. The special contexts E are *evaluation contexts* and have the following definition for the call-by-value and call-by-name variants of Λ respectively [27]:

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

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. Since, a call-by-name language never evaluates arguments, evaluation contexts do not include contexts of the shape $(V\ E_n)$.

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

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

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 function $\longmapsto$. The semantics of Λ is defined as follows:

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

For the definition of the semantics, η and η_v do not play any role. Their relevance for calculi is clarified in Section 1.2.3.

An important fact for the remainder of the thesis is that the syntax of the call-by-value language Λ can be redefined as follows:

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

The set of evaluation contexts has also the following equivalent definition:

$$E ::= [\,] \mid E[(V\ [\,])] \mid E[([\,]\ M)] \qquad (\mathit{EvCont})$$

We use all the definitions interchangeably. Moreover, we extend the notions of reductions to evaluation contexts by treating the hole as a place-holder for an arbitrary expression. For example, the reduction:

$$((\lambda x.(x\ y))\ [\,]) \longrightarrow ([\,]\ y)$$

is a β-reduction; it is *not* a β_v-reduction.

1.2.3 Observational and Denotational Equivalences

Not only do calculi define the semantics of Λ, but they are also useful for proving the correctness of some *optimizations*. Abstractly, an optimization of a program $C[M]$ is the replacement of M by a "more efficient" expression N such that a programmer cannot distinguish the observational behavior of the programs $C[M]$ and $C[N]$.

The formal definition of observational equivalence is parameterized by the set of relevant "observations" and by the set of relevant contexts $\mathcal{C}$. Without loss of generality, we will restrict ourselves to observing the termination behavior of programs. The notation $M \cong_v^{\mathcal{C}} N$ means that the two call-by-value Λ-expressions M and N are observationally equivalent relative to contexts in $\mathcal{C}$. We say that $M \cong_v^{\mathcal{C}} N$ if the following condition holds:

> For all contexts $C \in \mathcal{C}$ such that both $C[M]$ and $C[N]$ are programs, either both $eval_v(C[M])$ and $eval_v(C[N])$ are defined or both are undefined.

A similar definition holds for call-by-name Λ-expressions using the subscript n instead of v in the above definition.

For interesting languages, it is undecidable to determine whether two expressions are observationally equivalent. However, $\boldsymbol{\lambda}\beta_v$ and $\boldsymbol{\lambda}\beta$ are two typical (weak) examples of theories that are sound with respect to observational equivalence.

Theorem 1.1 *(Plotkin)* Let $M, N \in \Lambda$.

1. If $\boldsymbol{\lambda}\beta_v \vdash M = N$ then $M \cong_v^\Lambda N$.
2. If $\boldsymbol{\lambda}\beta \vdash M = N$ then $M \cong_n^\Lambda N$.

The soundness of extensions of $\boldsymbol{\lambda}\beta$ and $\boldsymbol{\lambda}\beta_v$ with η and η_v, respectively, depends on the circumstances. The axiom η_v is sound with respect to call-by-value observational equivalence for Λ. If we extend Λ with constants, η_v may be unsound. For an example, consider a dynamically typed language with numerals and a predicate *integer*?. The latter can distinguish 3 and $(\lambda x.(3\ x))$, yet, the η_v axiom identifies the two terms. In a typed setting, η_v is generally sound, independent of the parameter-passing technique.

The axiom η, on the other hand, fails to be sound with respect to call-by-name observational equivalence even in a pure language. For example, if Ω is a diverging term, then $(\lambda x.\Omega x)$ reduces to Ω but the two are clearly observationally distinct terms. Indeed, η is only sound in a typed language that does not permit the observation of the termination behavior of higher-type expressions.

If the terms $C[M]$ and $C[N]$ are simply typed Λ-programs, then the termination criterion is useless as the evaluation of any simply typed Λ-program terminates [6]. Thus to compare two simply typed expressions M and N, we enrich the set of contexts $\mathcal{C}$ to allow recursion and basic arithmetic operations. The canonical set of contexts is the set of PCF-contexts that includes numerals, an increment procedure **add1**, a decrement procedure **sub1**, conditionals, and a recursion operator [43]. Observational equivalence of call-by-name simply typed terms relative to PCF-contexts is decidable and axiomatized by the axioms $\beta\eta$.

Theorem 1.2 *(Meyer [81])* Let M and N be simply typed terms in Λ, then $\boldsymbol{\lambda}\beta\eta \vdash M = N$ if and only if $M \cong_n^{PCF} N$.

In general, it is hard to reason directly about observational equivalence. Denotational models provide alternative methods for reasoning about the observational equivalence relation. Intuitively, a denotational semantics associates each expression in the language with a domain element. Two expressions are *denotationally equivalent* if they denote the same element in the semantics. Denotational equivalence always implies observational equivalence but the converse is only true if the model is *fully abstract*. In the case of the simply typed λ-calculus, denotational equivalence in many familiar models reduces to $\beta\eta$ equivalence, *i.e.*, the theory $\boldsymbol{\lambda}\beta\eta$ proves exactly all the equations that are valid in those denotational models.

1.2.4 Manufacturing Equational Logics

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

> **Definition 1.1** *(Equational Correspondence)* Let $\mathcal{R}$ and $\mathcal{G}$ be two languages with calculi $\boldsymbol{\lambda}X_{\mathcal{R}}$ and $\boldsymbol{\lambda}X_{\mathcal{G}}$ respectively. Also let $f:\ \mathcal{R} \to \mathcal{G}$ be a translation from $\mathcal{R}$ to $\mathcal{G}$, and $h:\ \mathcal{G} \to \mathcal{R}$ be a translation from $\mathcal{G}$ to $\mathcal{R}$. Finally let $r, r_1, r_2 \in \mathcal{R}$ and $g, g_1, g_2 \in \mathcal{G}$. Then the calculus $\boldsymbol{\lambda}X_{\mathcal{R}}$ *equationally corresponds* to the calculus $\boldsymbol{\lambda}X_{\mathcal{G}}$ if the following four conditions hold:
>
> 1. $\boldsymbol{\lambda}X_{\mathcal{R}} \vdash r = (h \circ f)(r)$.
> 2. $\boldsymbol{\lambda}X_{\mathcal{G}} \vdash g = (f \circ h)(g)$.
> 3. $\boldsymbol{\lambda}X_{\mathcal{R}} \vdash r_1 = r_2$ if and only if $\boldsymbol{\lambda}X_{\mathcal{G}} \vdash f(r_1) = f(r_2)$.
> 4. $\boldsymbol{\lambda}X_{\mathcal{G}} \vdash g_1 = g_2$ if and only if $\boldsymbol{\lambda}X_{\mathcal{R}} \vdash h(g_1) = h(g_2)$.

The above correspondence is similar to the correspondence between the λ-calculus and combinatory logic [6, 18].

Usually, we will be given the two languages $\mathcal{R}$ and $\mathcal{G}$, the translation f, and the equational theory for $\mathcal{G}$. Our goal will be to derive the theory for $\mathcal{R}$ that equationally corresponds to the theory for $\mathcal{G}$. The derivation of the axioms $X_{\mathcal{R}}$ is systematic. For ease of reference, we outline the general procedure here.

The first step is to develop an *inverse translation* $h = f^{-1} : \mathcal{G} \to \mathcal{R}$, and find axioms $X^0_{\mathcal{R}}$ such that for all $r \in \mathcal{R}$:

$$\boldsymbol{\lambda}X^0_{\mathcal{R}} \vdash r = (f^{-1} \circ f)(r).$$

The inverse translation f^{-1} must be *compositional*, *i.e.*, for all contexts C and terms g in $\mathcal{G}$,

$$f^{-1}(C[g]) = f^{-1}(C[x])\,[x := f^{-1}(g)]\ ,$$

where x does not occur in C. The compositional nature of f^{-1} ensures that the equations resulting from the inversion process are valid in *any* context as required by equational logics in general. Naturally, the axioms $X^0_{\mathcal{R}}$ must be *sound* with respect to the translation f and $X_{\mathcal{G}}$:

$$\boldsymbol{\lambda}X^0_{\mathcal{R}} \vdash r_1 = r_2 \quad \text{implies} \quad \boldsymbol{\lambda}X_{\mathcal{G}} \vdash f(r_1) = f(r_2).$$

Next, for each equation $X^i_{\mathcal{G}} \in X_{\mathcal{G}}$ such that: $X^i_{\mathcal{G}} \stackrel{df}{=} (g_1 = g_2)$, we find a set of equational axioms $X^i_{\mathcal{R}}$ such that:

$$\boldsymbol{\lambda} X^i_{\mathcal{R}} \vdash f^{-1}(g_1) = f^{-1}(g_2).$$

The full equational theory for $\mathcal{R}$ then consists of the axioms $X_{\mathcal{R}}$ defined by:

$$X_{\mathcal{R}} \stackrel{df}{=} X^0_{\mathcal{R}} \cup \bigcup_i X^i_{\mathcal{R}}.$$

For the cases of interest to us, this equational theory is *sound* and *complete* with respect to the equational theory for $\mathcal{G}$. Technically, "soundness" refers to the left-to-right implication in Definition 1.1 and "completeness" refers to the right-to-left implication.

Chapter 2

A History of CPS Transformations

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

2.1 History and Motivation: Procedure Call versus Goto

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

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

2.1.1 Procedure Call as Goto

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

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

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

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

$$
\begin{array}{l}
((\textbf{lambda}\ (f)\ (f\ 1)) \\
\boxed{\begin{array}{l}(\textbf{let}\ ([y\ 2]) \\ \quad (\textbf{lambda}\ (x)\ (+\ y\ x)))\end{array}})
\end{array}
$$

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

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

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

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

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

2.1.2 Goto as Procedure Call

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

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

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

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

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

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

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

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

2.1.3 Continuations and Transfers of Control

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

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

2.2 The Original CPS Transformation

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

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

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

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

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

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

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

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

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

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

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

2.3 Compacting CPS Programs

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

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

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

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

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

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

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

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

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

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

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

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

Switching the order of arguments to the procedures yields:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

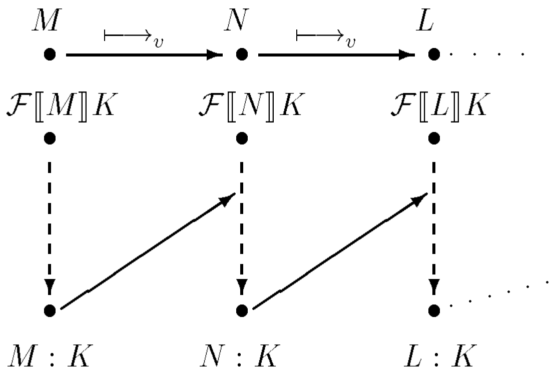
To complete the proof of Proposition 2.2, we show that if $\mathcal{F}[\![M]\!]$ reduces to two normal forms P and Q, then P and Q are identical.

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

Although $\mathcal{C}_k$, 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 2.5 *(Size)* The *size* of a term M (denoted by $|M|$) is the number of variables in M (including binding occurrences). The *size* of a context E (denoted by $|E|$) is the number of variables in E (including binding occurrences) plus 2.

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

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

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

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

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

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

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

The extension of the Fischer CPS transformation is the following:

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

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

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

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

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

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

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

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

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

There are two cases:

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

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

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

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

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

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

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

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

The result follows by the inductive hypothesis.

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

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

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

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

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

$\square$

2.5 Summary

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

Chapter 3

Equational Correspondence I: Functional Subset

The first advantage of CPS compilers over direct compilers is that the former may use the full theory of the λ-calculus to optimize programs while the latter usually rely on the weaker λ_v-calculus. This discrepancy naturally raises the following theoretical question: What additional call-by-value reductions are required to prove all the equations provable in the CPS framework? In the first section, we use the compacting CPS transformation developed in the previous chapter to identify two of the call-by-value reductions that correspond to the administrative CPS reductions. Next, we develop an "inverse" to the CPS transformation and use it to identify the remaining call-by-value reductions. The last two sections discuss the theoretical implications of our result and its connection to some of the open problems in the literature.

3.1 Administrative Source Reductions: The A-Reductions

In the previous chapter, we derived the compacting CPS transformation $\mathcal{C}_k$ from a two-phase process that first performs the Fischer CPS transformation and then performs all administrative reductions. In other words, our previous development corresponds to the right triangle of the following diagram:

$$\begin{tikzcd}
\bullet \arrow[r, "\mathcal{F}"] \arrow[d, dashed, "\text{Administrative Source Reductions}"'] \arrow[dr, "\mathcal{C}_k"] & \bullet \arrow[d, "\text{Administrative CPS Reductions}"] \\
\bullet \arrow[r, dashed] & \bullet
\end{tikzcd}$$

The diagram naturally suggests an alternative "decomposition" of the transformation $\mathcal{C}_k$. Instead of viewing administrative reductions as reductions on CPS terms, we perform the administrative reductions on source terms *before* conversion to CPS.

The definition of the function $\mathcal{C}_k$ naturally identifies two administrative source reductions:

1. As described in Section 2.4, the CPS transform of $E[((\lambda x.M)\ V)]$ lifts the redex outside the evaluation context producing $((\lambda x.E[M])\ V)$ and then converts the resulting term to CPS. In other words, the CPS transformation proper is conceptually preceded by the following administrative source reduction:

   $$E[((\lambda x.M)\ V)] \longrightarrow ((\lambda x.E[M])\ V).$$

   A similar reduction also applies to evaluation contexts $E[((\lambda x.M)\ [\ ])]$. Since the "hole" is a place-holder for an arbitrary term, the general form of the first administrative source reduction is:

   $$E[((\lambda x.M)\ N)] \ \longrightarrow\ ((\lambda x.E[M])\ N) \qquad (\beta_{\mathit{lift}})$$

   where $E \neq [\ ]$ and $x \notin FV(E)$.

2. The second administrative source reduction implicitly performed by the transformation $\mathcal{C}_k$ occurs during the transformation of evaluation contexts $E[([\ ]\ L)]$. As also explained in Section 2.4, the CPS transformation first names the result of the unknown application producing $E[((\lambda u.(u\ L))\ [\ ])]$ and then converts the resulting term to CPS. Thus, the second administrative reduction introduces new names for applications of unknown function values:

   $$((z\ M)\ L) \ \longrightarrow\ ((\lambda u.(u\ L))\ (z\ M)) \qquad (\beta_{\mathit{flat}})$$

   where $u \notin FV(L)$.

In the sequel, we will refer to these two reductions as the A-reductions.

Definition 3.1 *(A-reductions)* The administrative source reductions (A-reductions) are:

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

To verify that the A-reductions are indeed administrative reductions, we check that the function $\mathcal{C}_k$ maps the two sides of each reduction to the same CPS term, *i.e.*, that all members of the A-equivalence classes are mapped to the same CPS term. (Proposition 3.8 shows that β_{lift} and β_{flat} are in fact the *only* administrative reductions encoded in the transformation $\mathcal{C}_k$.)

Lemma 3.1 *(β_{lift}, β_{flat})* Let $M, N, L \in \Lambda$, $E \in EvCont(\Lambda)$, and $z \in Vars$:

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

Proof The proof of the first claim is straightforward. The identity in the second statement holds modulo the decorating over-lines above the administrative λ-abstractions. 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 over-line 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 $\mathcal{C}_k[\![((z\ E[(y\ V)])\ L)]\!]$

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

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

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

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 over-line does not create an administrative redex.

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

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

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

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

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

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

□

3.2 The CPS Language

Having identified the source reductions that correspond to administrative CPS reductions, we turn our attention to other $\beta\eta$-reductions on CPS terms. For this purpose, the relevant set of CPS terms is:

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

The context-free grammar that generates the set S can be directly derived from the right hand sides of the equations in Definition 2.4. 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.

Definition 3.2 *(CPS grammar, CPS program, cps(Λ))* Let $K\text{-}Vars = \{k\}$ be a set of continuation variables such that $Vars \cap K\text{-}Vars = \emptyset$.

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

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. The proof requires a new lexicographic measure for the induction.

Definition 3.3 *(Lexicographic Measure)* Let G be an element P of $cps(\Lambda)$ or an element K of $cps(EvCont)$, then the measure is $\langle \tilde{G}, |G| \rangle$ where $\tilde{G}$ is the number of abstractions of the form $\lambda k.K$ in G and $|G|$ is the size of G.

Theorem 3.2 The set S defined as $\{P \mid \exists M \in \Lambda.\ \boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow P\}$ is identical to the set $cps(\Lambda)$.

Proof For the left to right inclusion, it suffices to show that the output of $\mathcal{C}_k$ is a subset of $cps(\Lambda)$ which is obvious, and that the latter language is closed under $\beta\eta$-reductions. The latter claim follows from the Subject Reduction Lemma (Lemma 3.3).

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 $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] \longrightarrow\!\!\!\!\!\rightarrow P$. The proof is by lexicographic induction on the measure $\langle \tilde{G}, |G| \rangle$ in Definition 3.3 and 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 $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M_1]\!] \longrightarrow\!\!\!\!\!\rightarrow P_1$. 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.

□

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 3.3 *(Subject Reduction)* 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 $K_1 \in cps(EvCont)$ and $W_1 \in cps(Values)$. 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, $K_2 \in cps(EvCont)$ 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 sub-cases:
     - $\boldsymbol{\lambda}\beta\eta \vdash (\lambda x.P_1) \longrightarrow (\lambda x.P_2)$ 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.

$\square$

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 3.4 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):

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

3.3 The Inverse CPS 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 3.4 *($\mathcal{C}^{-1}, \Phi^{-1}, \mathcal{K}^{-1}$)* 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 3.5 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 measure in Definition 3.3:

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,

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

   (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,

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

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

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

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.
□

3.4 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 3.1). Our inverse CPS transformation maps this CPS term back to a particular element of the equivalence class, the element in A-normal form. In order to establish this latter result, we first define a subset of Λ in A-normal form.

Definition 3.5 *(Λ_a)* The language Λ_a is a subset of Λ that only includes terms in A-normal form.

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

We omit the simple inductive proof that the elements of the language are actually in A-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 A-normal form.

Lemma 3.6 Let $P \in \mathit{cps}(\Lambda)$ and $K \in \mathit{cps}(\mathit{EvCont})$, then $\mathcal{C}^{-1}[\![P]\!] \in \Lambda_a$ and $\mathcal{K}^{-1}[\![K]\!] \in \mathit{EvCont}_a$.

Proof The proof is by lexicographic induction on the measure in Definition 3.3. 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, $\mathcal{K}^{-1}[\![K]\!] \in \mathit{EvCont}_a$. It remains to establish that $\Phi^{-1}[\![W]\!] \in \mathit{Values}_a$.

   (a) $W = x$, then $\Phi^{-1}[\![W]\!] \equiv x \in \mathit{Values}_a$.

   (b) $W = \lambda k.k$, then $\Phi^{-1}[\![W]\!] \equiv \lambda x.x \in \mathit{Values}_a$.

   (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 $\Phi^{-1}[\![W]\!] \in \mathit{Values}_a$.

   (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.

$\square$

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 A-normal form. Naturally, if a term is already in A-normal form, then the composition yields the identity function.

Theorem 3.7 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 sub-cases:

   (a) $V = x$: then $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![x]\!] \equiv x$.

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

$$\begin{aligned} \lambda x.N &\longrightarrow\!\!\!\!\rightarrow \lambda x.(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![N]\!] && \text{(induction)} \\ &\equiv (\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![\lambda x.N]\!] \end{aligned}$$

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.*, $E[(x\ V)] \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E]\!][(x\ V)]$. The result follows because $V \longrightarrow\!\!\!\!\!\rightarrow (\Phi^{-1} \circ \Phi)[\![V]\!]$ 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,

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

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

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

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:

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

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:

$$E_1[(z\ (x\ L))] \longrightarrow\!\!\!\!\!\rightarrow (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1[(z\ [\ ])]]\!][(x\ L)] \equiv (\mathcal{K}^{-1} \circ \mathcal{K}_k)[\![E_1]\!][(z\ (x\ L))].$$

The result is immediate since the inductive hypothesis applies to E_1.

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

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

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

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

The proof of the second claim proceeds as above but is restricted to terms of the form $\mathcal{C}^{-1}[\![P]\!]$ for some $P \in \mathit{cps}(\Lambda)$. By the grammar in Definition 3.5, 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 $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$. □

Put differently, the theorem asserts that the reductions $\beta_{\mathit{lift}}\beta_{\mathit{flat}}$ 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 $\beta_{\mathit{lift}}\beta_{\mathit{flat}}$.

Proposition 3.8 If $\mathcal{C}_k[\![M]\!] \equiv \mathcal{C}_k[\![N]\!]$, then $\boldsymbol{\lambda}\beta_{\mathit{lift}}\beta_{\mathit{flat}} \vdash M = N$.

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 3.7, both M and N reduce to L by $\beta_{\mathit{lift}}\beta_{\mathit{flat}}$-reductions. It follows that $\boldsymbol{\lambda}\beta_{\mathit{lift}}\beta_{\mathit{flat}} \vdash M = N$. □

3.5 Complete Calculus for Pure Call-by-Value 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 $\boldsymbol{\lambda}\beta_v AB$ 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.

3.5.1 Completeness

As specified in Corollary 3.4, the possible β- and η-reductions on CPS terms are β_w, β_k, η_w, and η_k. To illustrate the derivation of the corresponding source reductions, we consider an η_k-reduction: $(\lambda x.K x) \longrightarrow K$ 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 AB includes all the previously derived reductions and η_v. See Figure 3.1 for a summary of these reductions.

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

Lemma 3.9 *(Completeness)* Let $P \in cps(\Lambda)$.

1. If $\boldsymbol{\lambda}\eta_k \vdash P \longrightarrow Q$ then $\boldsymbol{\lambda}\beta_v\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![P]\!] \twoheadrightarrow \mathcal{C}^{-1}[\![Q]\!]$.
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 $\boldsymbol{\lambda}\eta_v \vdash \mathcal{C}^{-1}[\![P]\!] \twoheadrightarrow \mathcal{C}^{-1}[\![Q]\!]$.

$$
\begin{array}{rcll}
((\lambda x.M)\ V) & \longrightarrow & M[x := V] & & (\beta_v) \\
(\lambda x.Vx) & \longrightarrow & V & x \notin FV(V) & (\eta_v) \\
E[((\lambda x.M)\ N)] & \longrightarrow & ((\lambda x.E[M])\ N) & x \notin FV(E), E \neq [\ ] & (\beta_{\mathit{lift}}) \\
((z\ M)\ L) & \longrightarrow & ((\lambda u.(u\ L))\ (z\ M)) & u \notin FV(L) & (\beta_{\mathit{flat}}) \\
((\lambda x.x)\ M) & \longrightarrow & M & & (\beta_{id}) \\
((\lambda x.E[(y\ x)])\ M) & \longrightarrow & E[(y\ M)] & x \notin FV(E[y]) & (\beta_\Omega)
\end{array}
$$

Figure 3.1 The Reductions $AB \stackrel{df}{=} \{\eta_v, \beta_{\mathit{lift}}, \beta_{\mathit{flat}}, \beta_{id}, \beta_\Omega\}$ for Λ.

4. If $\boldsymbol{\lambda}\beta_w \vdash P \longrightarrow Q$ then $\boldsymbol{\lambda}\beta_v\beta_{\mathit{lift}}\beta_{id}\beta_\Omega \vdash \mathcal{C}^{-1}[\![P]\!] \twoheadrightarrow \mathcal{C}^{-1}[\![Q]\!]$.

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: By Lemma 3.10.

□

Lemma 3.10 Let $P \in cps(\Lambda)$, and $W \in cps(\mathit{Values})$, then:

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

Proof We have:

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

The first three steps are straightforward; for the last step, we need to 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 the measure in Definition 3.3 and proceeds by cases on the arguments to $\mathcal{C}^{-1}$ and $\mathcal{K}^{-1}$:

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

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

   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 $\Phi^{-1}[\![W_1]\!][x := \Phi^{-1}[\![W]\!]]$

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

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

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

2. $K = k$: then the claim is vacuously true because $k \neq x$.

3. $K = ((\lambda k.K_1)\ K_2)$: then $\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!][x := \Phi^{-1}[\![W]\!]]$

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

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

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

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 $\mathcal{K}^{-1}[\![(x\ K_1)]\!][x := \Phi^{-1}[\![W]\!]]$

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

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) $W = \lambda k.W_3K$: then, $\mathcal{K}^{-1}[\![K']\!][((\lambda y.\mathcal{C}^{-1}[\![((W_3K)\ y)]\!])\ [\ ])]$

$$\begin{array}{lll} \equiv & \mathcal{K}^{-1}[\![K']\!][\mathcal{K}^{-1}[\![(\lambda y.((W_3K)\ y))]\!]] & \\ \twoheadrightarrow & \mathcal{K}^{-1}[\![K']\!][\mathcal{K}^{-1}[\![(W_3K)]\!]] & \text{(case 1 of Lemma 3.9)} \\ \twoheadrightarrow & \mathcal{K}^{-1}[\![(W_3K)[k := K']]\!] & \text{(Auxiliary Claim)} \\ \equiv & \mathcal{K}^{-1}[\![((\lambda k.W_3K)\ K')]\!]. & \end{array}$$

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

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

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

1. $\boldsymbol{\lambda}\beta_{\mathit{lift}} \vdash \mathcal{K}^{-1}[\![K]\!][\mathcal{C}^{-1}[\![P]\!]] \twoheadrightarrow \mathcal{C}^{-1}[\![P[k := K]]\!]$

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

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 $\mathcal{K}^{-1}[\![K]\!][\mathcal{C}^{-1}[\![P]\!]]$

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

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 $\mathcal{K}^{-1}[\![K_2]\!][\mathcal{K}^{-1}[\![K_3]\!][(x\ [\ ])]]$

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

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

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

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

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

□

The completeness theorem is a direct consequence of the above results.

Theorem 3.11 *(Completeness)* Let $P \in cps(\Lambda)$. If $\boldsymbol{\lambda}\beta\eta \vdash P \longrightarrow\!\!\!\!\!\rightarrow Q$ then $\boldsymbol{\lambda}\beta_v AB \vdash \mathcal{C}^{-1}[\![P]\!] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}^{-1}[\![Q]\!]$.

Proof By pasting together the proofs of the completeness lemma. □

3.5.2 Soundness

The set of source reductions in Figure 3.1 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 3.12 *(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 3.1. We prove the statement for β_v-reductions. The other proofs are similar.

First, we state the following result without proof.

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

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. $\boldsymbol{\lambda}\beta \vdash \mathcal{C}_k[\![M]\!][x := \Phi[\![V]\!]] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{C}_k[\![M[x := V]]\!]$.

2. $\boldsymbol{\lambda}\beta \vdash \mathcal{K}_k[\![E]\!][x := \Phi[\![V]\!]] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{K}_k[\![E[x := V]]\!]$.

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 $\mathcal{C}_k[\![E[(x\ U)]]\!][x := \Phi[\![V]\!]]$

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

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:

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

We proceed by cases of V:

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

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

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

□

The soundness theorem summarizes the results of this subsection.

Theorem 3.13 *(Soundness)*
If $\boldsymbol{\lambda}\beta_v AB \vdash M \longrightarrow\!\!\!\!\!\rightarrow N$ 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. □

3.5.3 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 3.5 and 3.7 about the composition of the CPS transformation and its inverse, they imply the results of Figure 3.2. In the figure, the dotted lines correspond to the application of

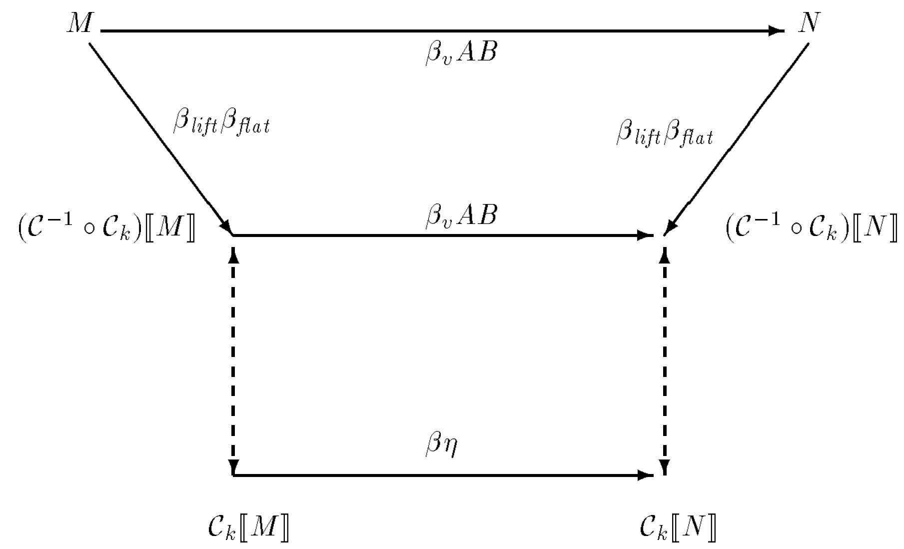


Figure 3.2 Reduction Steps in Λ versus $cps(\Lambda)$.

$\mathcal{C}_k$ 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.

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

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

For such transformations, it is straightforward to deduce variants of Theorems 3.5, 3.7, 3.11, and 3.13 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 1.1.

Theorem 3.14 *(Equational Correspondence)* The theory $\boldsymbol{\lambda}\beta_v AB$ equationally corresponds to the theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages Λ and $cps(\Lambda)$, and the translations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

The formulation of the calculus $\boldsymbol{\lambda}\beta_v AB$ based on the six reductions in Figure 3.1 is only necessary for the correspondence of reduction steps. For the equational correspondence, we summarize the reductions using the equational axioms with the same name in Figure 3.3.

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

Figure 3.3 The Axioms $AB \stackrel{df}{=} \{\eta_v, \beta_{\mathit{lift}}, \beta'_\Omega\}$ for Λ.

3.6 Corollaries

The development of a calculus for call-by-value languages that is complete with respect to $\beta\eta$-equality on CPS programs has some important theoretical consequences. The importance stems from two factors. First, the calculus $\boldsymbol{\lambda}\beta\eta$ is a canonical and complete system of reasoning about functions in a call-by-name language. Second, our newly developed calculus turns out to be equivalent to Moggi's computational λ-calculus, which is a generic calculus for reasoning about programs with computational effects [67, 68]. Our correspondence theorem relates these two canonical calculi and hints at deeper connections between continuations and other computational effects.

In the first subsection, we outline the equivalence of the calculus $\boldsymbol{\lambda}\beta_v AB$ and Moggi's computational λ-calculus. Next, we show the completeness of a typed version of the calculus $\boldsymbol{\lambda}\beta_v AB$ with respect to denotational equivalence in conventional

CPS models. Finally, we discuss the relationship of our calculus to the call-by-value observational equivalence relation.

3.6.1 Moggi's Computational Lambda Calculus

Starting from a categorical semantics of computations, Moggi [67] introduced the computational λ-calculus as a basis for reasoning about programs, independently of any specific computational model. In the Edinburgh LFCS Technical Report, Moggi formulates an untyped variant of his calculus $\boldsymbol{\lambda}_c$ using the following reductions:

$$
\begin{array}{rcll}
((\lambda x.M)\ V) & \longrightarrow & M[x := V] & (\beta_v) \\
(\lambda x.Vx) & \longrightarrow & V \qquad x \notin FV(V) & (\eta_v) \\
(\mathbf{let}\ (x\ M)\ x) & \longrightarrow & M \qquad x \notin FV(M) & (id) \\
(\mathbf{let}\ (x_2\ (\mathbf{let}\ (x_1\ M_1)\ M_2))\ M) & \longrightarrow & (\mathbf{let}\ (x_1\ M_1)\ (\mathbf{let}\ (x_2\ M_2)\ M)) & (Comp) \\
(\mathbf{let}\ (x\ V)\ M) & \longrightarrow & M[x := V] & (let_v) \\
((M\ N)\ L) & \longrightarrow & (\mathbf{let}\ (x\ (M\ N))\ (x\ L)) & (let.1) \\
(V\ (M\ N)) & \longrightarrow & (\mathbf{let}\ (x\ (M\ N))\ (V\ x)) & (let.2)
\end{array}
$$

It is straightforward to show that the resulting calculus is equivalent to our extension of the call-by-value λ-calculus, *i.e.*, is equivalent to the calculus $\boldsymbol{\lambda}\beta_v AB$. Formally, the equivalence of the two calculi means that each calculus can prove the equations of the other. We outline the proof below.

For the purposes of the proof, we identify **let**-expressions with their expansions as applications of λ-expressions:

$$(\mathbf{let}\ (x\ M)\ N) \equiv ((\lambda x.N)\ M)$$

Using the above identity, the two calculi have the following rules in common: β_v, let_v, η_v, β_{id}, and id. To show one direction of the equivalence, we note that $Comp$ is an instance of β_{lift} and that $let.1$ an instance of β_Ω. Finally, $let.2$ is simply an η_v expansion. For the reverse direction, it is sufficient to show that $\boldsymbol{\lambda}_c$ proves β_{lift} and β_Ω for the cases when E is empty, $E \equiv (V\ [\,])$, and $E \equiv ([\,]\ M)$, which is straightforward.

Based on the above argument, the correspondence of the calculi yields the following reformulation of the Correspondence theorem.

> **Theorem 3.15** (Correspondence (Reformulation)) The theory $\boldsymbol{\lambda}_c$ equationally corresponds to the theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages Λ and $cps(\Lambda)$, and the translations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

As also noted by Filinski [35, 36], the correspondence between Moggi's computational λ-calculus (a generic calculus for computational effects) and the calculus $\boldsymbol{\lambda}\beta_v AB$ (a calculus derived from the CPS transformation) is but one aspect of a deeper correspondence between continuations and other computational effects. At this point, the relationship between our result and Filinski's is superficial and a deeper investigation is needed.

3.6.2 Denotational CPS Models

The calculus $\boldsymbol{\lambda}\beta\eta$ is a canonical and complete system for reasoning about functions in a simply typed call-by-name language. It is therefore important to determine whether the equational correspondence proved in this chapter holds in a typed setting. A positive answer would imply that the calculus $\boldsymbol{\lambda}\beta_v AB$ is canonical and complete with respect to equivalence in a certain class of denotational models.

We begin by defining the typed variants of the source language, the CPS language, and the CPS transformation. For the source language, we define the following syntactic category of types:

$$t ::= o \mid t \rightarrow t$$

where o refers to an *observable* base type, *e.g.*, natural numbers. The term language is the subset of Λ to which we can assign simple types as follows:

$$M \ ::= \quad x^t \mid (M^{s \rightarrow t}\ N^s)^t \mid (\lambda x^s . M^t)^{s \rightarrow t}$$

The CPS transformation on simply typed terms can be factored into a CPS transformation on types and a CPS transformation on raw (untyped) terms [65]. If M is of type t, then $\mathcal{F}[\![M]\!]$ is of type $(\overline{t} \rightarrow a) \rightarrow a$ where a is a distinguished type of *answers*:[5]

$$\begin{aligned} \overline{o} &= o \\ \overline{t \rightarrow s} &= (\overline{s} \rightarrow a) \rightarrow \overline{t} \rightarrow a \end{aligned}$$

The set of CPS terms is the subset of the language $cps(\Lambda)$ for which we can assign simple types.

[5]Assuming that the continuation is the first argument to a procedure. Switching the order of arguments so that the continuation becomes the second argument, we get:

$$\begin{aligned} \overline{o} &= o \\ \overline{t \rightarrow s} &= \overline{t} \rightarrow (\overline{s} \rightarrow a) \rightarrow a \end{aligned}$$

Definition 3.6 *($cps(\Lambda^t)$)* The simply typed CPS terms are:

$$\begin{array}{rcl} P & ::= & (K^{t\rightarrow a}\ W^t)^a \\ W & ::= & x^t \mid (\lambda k^{s\rightarrow a}.K^{t\rightarrow a})^{(s\rightarrow a)\rightarrow t\rightarrow a} \\ K & ::= & k^{t\rightarrow a} \mid (W^{(s\rightarrow a)\rightarrow t\rightarrow a}\ K^{s\rightarrow a})^{t\rightarrow a} \mid (\lambda x^t.P^a)^{t\rightarrow a} \end{array}$$

To verify that the correspondence theorem holds for typed languages, it suffices to check that both the CPS transformation and its inverse preserve the typability of terms.

Lemma 3.16 Let $M \in \Lambda^t$, and $P \in cps(\Lambda^t)$, then $\mathcal{C}_k[\![M]\!] \in cps(\Lambda^t)$ and $\mathcal{C}^{-1}[\![P]\!] \in \Lambda^t$.

As a corollary, Theorem 3.14 extends to simply typed languages.

Theorem 3.17 *(Correspondence for Typed Languages)* The typed theory $\boldsymbol{\lambda}\beta_v AB$ equationally corresponds to the typed theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages Λ^t and $cps(\Lambda^t)$, and the translations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

The above correspondence establishes a connection between two syntactic theories; this connection can now be used to establish the soundness and completeness of the computational λ-calculus with respect to the conventional CPS denotational model. The result explained below follows from the soundness and completeness of the simply typed λ-calculus with respect to the full type structure [42].

In the conventional semantics for Λ^t [42, 47], each type denotes a set of elements: the base type o denotes some arbitrary infinite set B, and the type $t_1 \rightarrow t_2$ denotes the set of functions from the denotations of t_1 to the denotations of t_2. We define the full type pre-structure over B as the collection of nonempty sets D^t for each type t:

$$\begin{array}{rcl} D^o & = & B \\ D^{t_1\rightarrow t_2} & = & (D^{t_1} \Rightarrow D^{t_2}) \end{array}$$

The full type structure $\mathcal{P}_B$ over B consists of a full type pre-structure and a *meaning* function $\mathcal{P}$ such that:

$$\begin{array}{rcl} \mathit{Env} : \mathit{Vars} & \Rightarrow & \cup_t D^t \qquad \text{such that for } \rho \in \mathit{Env},\ \rho(x^t) \in D^t \\ \mathcal{P}[\![x]\!]\rho & = & \rho(x) \\ \mathcal{P}[\![\lambda x.M]\!]\rho & = & \underline{\lambda} a.\mathcal{P}[\![M]\!]\rho[x/a] \\ \mathcal{P}[\![MN]\!]\rho & = & \mathcal{P}[\![M]\!]\rho(\mathcal{P}[\![N]\!]\rho) \end{array}$$

The notation $\underline{\lambda}a.\cdots$ denotes the function f such that $f(a) = \cdots$. The notation $\mathcal{P}_B \models M = N$ means that for all environments ρ, $\mathcal{P}[\![M]\!]\rho = \mathcal{P}[\![N]\!]\rho$. Based on this definition, we can formulate the completeness theorem.

Theorem 3.18 *(Friedman [42])* Let M, N be simply typed terms in Λ. Then, $\boldsymbol{\lambda}\beta\eta \vdash M = N$ if and only if $\mathcal{P}_B \models M = N$.

By analogy, the CPS type structure $\mathcal{S}_B$ over some infinite base set B and an infinite set of answers A, consists of a nonempty set D^s for each type and a *meaning* function such that:

$$
\begin{array}{rcl}
D^o & = & B \\
D^{(s_1 \to a) \to s_2 \to a} & = & ((D^{s_1} \Rightarrow A) \Rightarrow D^{s_2} \Rightarrow A)
\end{array}
$$

and

$$
\begin{array}{rcl}
\mathcal{S} : \Lambda \times \mathit{Env} \times \mathit{Continuation} & \Rightarrow & A \\
\mathit{Env} : \mathit{Vars} & \Rightarrow & \cup_s D^s \qquad \text{such that for } \rho \in \mathit{Env},\ \rho(x^s) \in D^s \\
\mathit{Continuation} : \cup_s D^s & \Rightarrow & A \\
\mathcal{S}[\![x]\!]\rho\kappa & = & \kappa(\rho(x)) \\
\mathcal{S}[\![\lambda x.M]\!]\rho\kappa & = & \kappa\ \underline{\lambda}c.\underline{\lambda}a.\mathcal{S}[\![M]\!]\rho[x/a]c \\
\mathcal{S}[\![MN]\!]\rho\kappa & = & \mathcal{S}[\![M]\!]\rho(\underline{\lambda}a.\mathcal{S}[\![N]\!]\rho(\underline{\lambda}b.((a\ \kappa)\ b)))
\end{array}
$$

Not surprisingly, the meaning of a term M in the CPS model is directly related to the meaning of $\mathcal{F}[\![M]\!]$ in the direct model.

Lemma 3.19 Let $M \in \Lambda$. $(\mathcal{P}[\![\mathcal{F}[\![M]\!]]\!]\rho\ \kappa) = \mathcal{S}[\![M]\!]\rho\kappa$.

Proof Idea The proof is by induction on M. $\square$

This lemma implies that $\mathcal{S}_B$ and $\mathcal{P}_B$ satisfy theorems that are related via the Fischer CPS transformation.

Proposition 3.20 Let $M, N \in \Lambda$. Then, $\mathcal{P}_B \models \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$ if and only if $\mathcal{S}_B \models M = N$.

Proof The proposition follows from Lemma 3.19. $\square$

It follows that the $\boldsymbol{\lambda}_c$-calculus is sound and complete with respect to the CPS type structure.

Theorem 3.21 Let $M, N \in \Lambda$. Then, $\boldsymbol{\lambda}_c \vdash M = N$ if and only if $\mathcal{S}_B \models M = N$.

Proof By the correspondence theorem, $\boldsymbol{\lambda}_c \vdash M = N$ if and only if $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$. By Friedman's completeness theorem, this is equivalent to $\mathcal{P}_B \models \mathcal{F}[\![M]\!] = \mathcal{F}[\![N]\!]$. The result follows by Proposition 3.20. $\square$

3.6.3 Observational Equivalence

The interest in calculi is generally motivated by their soundness with respect to observational equivalence. Therefore, the natural question is whether our extension is sound with respect to the call-by-value observational equivalence relation. Moggi [67] proved the result in a typed setting.

In a dynamically typed language, the soundness of the $\boldsymbol{\lambda}_c$-calculus with respect to the call-by-value observational equivalence relation depends on the particular language extensions. For example, the axiom η_v is *unsound* in languages like Lisp or Scheme. It is still possible to prove the soundness of the $\boldsymbol{\lambda}_c$-calculus for pure dynamically typed languages, i.e., languages with no constants.

Theorem 3.22 Let $M, N \in \Lambda$. If $\boldsymbol{\lambda}\beta_v AB \vdash M = N$ then $M \cong_v^{\Lambda} N$.

Proof Let C be a context such that $C[M], C[N]$ are closed Λ-terms. Assume $\boldsymbol{\lambda}\beta_v AB \vdash M = N$ and $eval_v(C[M])$ is defined. The goal is to show that $eval_v(C[N])$ is also defined.

It follows from the assumptions that $\boldsymbol{\lambda}\beta_v AB \vdash C[M] = C[N]$ by *Compatibility*. Therefore,

$$\boldsymbol{\lambda}\beta\eta \ \vdash \ \mathcal{C}_k[\![C[M]]\!] = \mathcal{C}_k[\![C[N]]\!] \tag{1}$$

by Theorem 3.14. Also by the assumptions and the definition of $eval_v$, $\boldsymbol{\lambda}\beta_v \vdash C[M] = V$ for some value V. Hence,

$$\boldsymbol{\lambda}\beta\eta \ \vdash \ \mathcal{C}_k[\![C[M]]\!] = \mathcal{C}_k[\![V]\!] \tag{2}$$

by Theorem 3.14. From (1) and (2), we deduce that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![C[N]]\!] = \mathcal{C}_k[\![V]\!] = (k\ \Phi[\![V]\!])$. The Church-Rosser theorem implies the existence of a term P such that:

$$\begin{aligned} \boldsymbol{\lambda}\beta\eta &\vdash\ \mathcal{C}_k[\![C[N]]\!] \twoheadrightarrow P \\ \boldsymbol{\lambda}\beta\eta &\vdash\ (k\ \Phi[\![V]\!]) \twoheadrightarrow P \end{aligned}$$

Obviously, all the reductions starting from the term $(k\ \Phi[\![V]\!])$ must occur inside $\Phi[\![V]\!]$. Since reductions preserve the syntactic categories in the CPS language, P must be of the form $(k\ W)$ for some W. Therefore, $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![C[N]]\!] \twoheadrightarrow (k\ W)$, and hence

$$\boldsymbol{\lambda}\beta\eta\ \vdash\ \mathcal{C}_k[\![C[N]]\!][k := \lambda x.x] \twoheadrightarrow ((\lambda x.x)\ W) \longrightarrow W$$

Lemma 3.23 implies that $eval_n((\mathcal{F}[\![C[N]]\!]\ \lambda x.x))$ is defined. Thus, $eval_v(C[N])$ is defined by Theorem 2.1. $\square$

Lemma 3.23 Let M be a closed Λ-term. If $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!][k := \lambda x.x] \twoheadrightarrow W$, then $eval_n(\mathcal{F}[\![M]\!]\ \lambda x.x)$ is defined.

Proof Assume $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!][k := \lambda x.x] \twoheadrightarrow W$, then by Lemma 2.5,

$$\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{F}[\![M]\!]\ \lambda x.x) \twoheadrightarrow ((\lambda k.\mathcal{C}_k[\![M]\!])\ \lambda x.x) \longrightarrow \mathcal{C}_k[\![M]\!][k := \lambda x.x] \twoheadrightarrow W$$

By the Postponement Lemma [6, p 15], we also have:

$$(\mathcal{F}[\![M]\!]\ \lambda x.x) \twoheadrightarrow_\beta L \twoheadrightarrow_\eta W \qquad \text{for some term } L$$

Since M is a closed term, W cannot be a variable; it must a λ-abstraction. Any η-*expansion* starting from a λ-abstraction will also result in a λ-abstraction. Therefore, L is a value:

$$(\mathcal{F}[\![M]\!]\ \lambda x.x) \twoheadrightarrow_\beta W'$$

By the Standard Reduction theorem [75], if a term reduces to a value, then it standard-reduces to a value. Therefore, $eval_n(\mathcal{F}[\![M]\!]\ \lambda x.x)$ is defined. $\square$

At this point, we have an extended call-by-value calculus that corresponds to a call-by-name calculus and each calculus is sound with respect to the appropriate notion of observational equivalence. The two operational equivalences however do not coincide, *e.g.*, $M \cong_v^\Lambda N$ does *not* imply that $\mathcal{C}_k[\![M]\!] \cong_n^\Lambda \mathcal{C}_k[\![N]\!]$. For example, if:

$$\begin{aligned} M &\stackrel{df}{=} \lambda y.\lambda x.x\ (y\ x) \\ N &\stackrel{df}{=} \lambda y.\lambda x.x\ (y\ \lambda z.xz) \end{aligned}$$

then, $M \cong_v^\Lambda N$ [75]. On the other hand,

$$\begin{aligned} \mathcal{C}_k[\![M]\!] &= (k\ \lambda k.\lambda y.(k\ \lambda k.\lambda x.((y\ (x\ k))\ x))), \\ \mathcal{C}_k[\![N]\!] &= (k\ \lambda k.\lambda y.(k\ \lambda k.\lambda x.((y\ (x\ k))\ (\lambda k.\lambda z.((x\ k)\ z))))), \end{aligned}$$

and the context:

$$\begin{aligned} D \stackrel{df}{=} &((\lambda k.[\,]) \\ &\quad (\lambda a.((a\ (\lambda b.((b\ (\lambda x.x))\ \lambda d.\Omega))) \\ &\qquad (\lambda k.\lambda m.m\ (\lambda x.x))))) \end{aligned}$$

differentiates the two expressions. Since D includes a sub-term $(\lambda k.\lambda m.(m\ \lambda x.x))$ that ignores its continuation, there is *no* context $C \in \Lambda$ such that $\mathcal{C}_k[\![C[M]]\!] = D[\mathcal{C}_k[\![M]\!]]$.

This result prompted Meyer and Riecke [64] to deduce that "continuations may be unreasonable." However, a restriction of D to range over contexts in the language $cps(\Lambda)$ results in a notion of observational equivalence that coincides with the call-by-value observational equivalence.

Theorem 3.24 Let $M, N \in \Lambda$.
Then, $M \cong_v N$ if and only if $\mathcal{C}_k[\![M]\!] \cong_n^{cps(\Lambda)} \mathcal{C}_k[\![N]\!]$

Proof The left-to-right implication is straightforward. For the reverse implication, assume $\mathcal{C}_k[\![M]\!] \not\cong_n^{cps(\Lambda)} \mathcal{C}_k[\![N]\!]$, then there exists a context D such that $D[\mathcal{C}_k[\![M]\!]]$, $D[\mathcal{C}_k[\![N]\!]] \in cps(\Lambda)$ and the evaluation of one of the programs terminates while the other diverges. Without loss of generality, assume $eval_n((\lambda k.D[\mathcal{C}_k[\![M]\!]])\ \lambda x.x)$ is defined and $eval_n((\lambda k.D[\mathcal{C}_k[\![N]\!]])\ \lambda x.x)$ is undefined. By the definition of $eval_n$, we get:

$$\begin{aligned} &\boldsymbol{\lambda}\beta \vdash ((\lambda k.D[\mathcal{C}_k[\![M]\!]])\ \lambda x.x) = W \\ &\boldsymbol{\lambda}\beta \nvdash ((\lambda k.D[\mathcal{C}_k[\![N]\!]])\ \lambda x.x) = W' \qquad \text{for any } W' \end{aligned}$$

By Lemma 3.25, there exists a context $C \in \Lambda$ such that:

$$\begin{aligned} &\boldsymbol{\lambda}\beta\eta \vdash ((\lambda k.\mathcal{C}_k[\![C[M]]\!])\ \lambda x.x) = W \\ &\boldsymbol{\lambda}\beta\eta \nvdash ((\lambda k.\mathcal{C}_k[\![C[N]]\!])\ \lambda x.x) = W' \qquad \text{for any } W' \end{aligned}$$

By Lemma 3.23 and Theorem 2.1, $eval_v(C[M])$ is defined and $eval_v(C[N])$ is undefined. Therefore, $M \not\cong_v^\Lambda N$. □

Lemma 3.25 Let $M, N \in \Lambda$, $D[\mathcal{C}_k[\![M]\!]], D[\mathcal{C}_k[\![N]\!]] \in cps(\Lambda)$. Then, there exists a context C such that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![C[M]]\!] = D[\mathcal{C}_k[\![M]\!]]$ and $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![C[N]]\!] = D[\mathcal{C}_k[\![N]\!]]$.

Proof Since $D[\mathcal{C}_k[\![M]\!]], D[\mathcal{C}_k[\![N]\!]] \in cps(\Lambda)$, they are valid arguments to $\mathcal{C}^{-1}$. It suffices to show that the function $\mathcal{C}^{-1}$ is a homomorphism when restricted to inputs of the form $\mathcal{C}_k[\![M]\!]$ for some $M \in \Lambda$. By cases:

1. $\mathcal{C}^{-1}[\![(K\ W)]\!] = \mathcal{K}^{-1}[\![K]\!][\Phi^{-1}[\![W]\!]]$.
2. $\Phi^{-1}[\![x]\!] = x$.
3. $\Phi^{-1}[\![(\lambda k.k)]\!] = \lambda x.x$.
4. $\Phi^{-1}[\![(\lambda k.WK)]\!] = \lambda x.\mathcal{C}^{-1}[\![(W\ K)\ x]\!]$.
5. $\Phi^{-1}[\![(\lambda k.\lambda x.P)]\!] = \lambda x.\mathcal{C}^{-1}[\![P]\!]$.
6. $\mathcal{K}^{-1}[\![k]\!] = [\ ]$.
7. $\mathcal{K}^{-1}[\![(x\ K)]\!] = \mathcal{K}^{-1}[\![K]\!][(x\ [\ ])]$.
8. $\mathcal{K}^{-1}[\![((\lambda k.K_1)\ K_2)]\!] = \mathcal{K}^{-1}[\![K_1[k := K_2]]\!]$.
9. $\mathcal{K}^{-1}[\![(\lambda x.P)]\!] = ((\lambda x.\mathcal{C}^{-1}[\![P]\!])\ [\ ])$.

The function is homomorphic is all cases except cases 4 and 8. Both are impossible if the input is of the form $\mathcal{C}_k[\![M]\!]$ for some $M \in \Lambda$. □

3.7 Related Work

Most of the material in this chapter is contained in the technical report version of our 1992 Lisp and Functional Programming paper [84]. Some of the ideas are closely related to recent work by Danvy, Lawall, and Hatcliff.

Danvy and Lawall [20, 22] studied the problem of inverting the CPS transformation and produced a "direct style" transformation similar to our transformation $\mathcal{C}^{-1}$. Danvy and Lawall argue that the "direct style" transformation is useful in its own right and discuss applications in partial evaluation. Their analysis did not however address the interactions among the CPS transformation, its inverse, reductions on call-by-value terms, and reductions on CPS terms.

In subsequent work, Danvy and Lawall [19, 57] recognize that the CPS transformation can be "staged" in a number of independent steps. Their staging is similar to ours; their first two stages essentially perform an A-normalization as explained in Section 3.1.

Finally, Danvy and Hatcliff [48] extended our correspondence theorems to languages with other evaluation strategies, *e.g.*, call-by-name.

3.8 Summary

We have identified the call-by-value reductions that can prove any equation that $\beta\eta$-reductions can prove on CPS terms. Our new reductions include a strongly normalizing subset, the A-reductions, which can be thought of as a first stage of the CPS transformation. The full set of reductions coincides with Moggi's computational λ-calculus, which hints at a deep connection between continuations and other computational effects. Our newly developed calculus is sound with respect to the pure call-by-value observational equivalence, and the typed version is complete with respect to equivalence in CPS denotational models. In the process of discovering the calculus, we have developed a new inverse CPS transformation, which is a useful tool in its own right. We have also analyzed the precise relationship between our optimizing CPS transformation and the inverse CPS transformation.

Chapter 4

Equational Correspondence II: Adding Effects

The correspondence theorems in the previous chapter are the theoretical foundation for similar correspondence theorems in realistic programming languages. Since continuations were conceived to explain sophisticated control operators [63, 69, 77, 101], we begin by extending the theorems to languages with such constructs. In the next section, we axiomatize the semantics of a language with assignments using the store-passing transformation. In Section 3, we combine functions, control operators, and assignments with other constructs necessary for realistic programming, and prove an appropriate version of the correspondence theorem. Section 4 includes an example showing the newly developed theory at work and Section 5 discusses the theoretical ramifications of our results.

4.1 Non-Local Control Operators

We first investigate the addition of the control operators *abort* ($\mathcal{A}$) and *call-with-current-continuation* (*callcc*). These operators suffice to express a wide variety of control abstractions such as error exits, jumps, backtracking, coroutines, and exception handling [40, 49]. The last two sections extend the discussion to include control delimiters as well.

Informally, $\mathcal{A}$ permits the program 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 program with a procedure-like 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 the previous chapter 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 equations of one calculus to the equations of the other. At this point, it is an open question whether the results can be re-established for reductions (as opposed to equalities).

4.1.1 The Extended Language and its Semantics

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

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

Instead of providing a formal semantics for *callcc* and $\mathcal{A}$ in terms of standard reductions, we follow the traditional route and immediately specify the translation of these values into CPS form and use this translation as the formal semantics of the language.[6] The extensions to $\mathcal{C}_k$ (or $\mathcal{F}$) consist of two additional clauses to the function Φ (or $\mathcal{F}_v$) [14]:

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

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:

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

In order to simplify the following discussions (and proofs), we use a CPS transformation that is less compacting than the one in Definition 2.4 but more suited for the remainder of our analysis.

[6] Felleisen *et al.* [27, 29, 30] and Talcott [103] use alternative definitions that do not rely on the CPS transformation.

Definition 4.1 *($\mathcal{C}_k$ with Control Operators)* Let $k, u_i \in \mathit{Vars}$ be variables that do not occur in the argument to $\mathcal{C}_k$.

$$\begin{aligned}
\mathcal{C}_k[\![V]\!] &= (k\ \Phi[\![V]\!]) \\
\mathcal{C}_k[\![E[(V_1\ V_2)]]\!] &= ((\Phi[\![V_1]\!]\ \mathcal{K}_k[\![E]\!])\ \Phi[\![V_2]\!])
\end{aligned}$$

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

$$\begin{aligned}
\mathcal{K}_k[\![[\,]]\!] &= k \\
\mathcal{K}_k[\![E[(V\ [\,])]]\!] &= (\Phi[\![V]\!]\ \mathcal{K}_k[\![E]\!]) \\
\mathcal{K}_k[\![E[([\,]\ M)]]\!] &= \lambda f.\mathcal{C}_k[\![E[(f\ M)]]\!]
\end{aligned}$$

Besides the extensions to *callcc* and $\mathcal{A}$, the transformation differs from the function in Definition 2.4 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.

4.1.2 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 3.2.

Definition 4.2 *(CPS grammar $cps(\Lambda + \mathit{callcc} + \mathcal{A})$)* The extended CPS terms are generated by the following grammar:

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

$x \in \mathit{Vars}$
$k \in K\text{-}\mathit{Vars} = \{k_1, k_2, \ldots\}$ and $K\text{-}\mathit{Vars} \cap \mathit{Vars} = \emptyset$

A comparison with Definition 3.2 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 $\Lambda + \mathit{callcc} + \mathcal{A}$ is the following.

Definition 4.3 Let $P \in \mathit{cps}(\Lambda + \mathit{callcc} + \mathcal{A})$. Let W and K be values and continuations in the same language:

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

The transformation differs from the function in Definition 3.4 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 [22].[7]

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 AB for the pure language and control specific axioms.

Definition 4.4 *(The Axioms C)*

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

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. The operator $\mathcal{A}$ eliminates evaluation contexts.

The relationship of $\mathcal{C}_k$ to the function $\mathcal{C}^{-1}$ is the subject of the following 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. 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 4.1 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. $\boldsymbol{\lambda}\beta\eta \vdash (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!][k_1 := (\lambda d.k_1), \ldots, k_n := (\lambda d.k_n)] = P$.

[7]Danvy and Lawall [22] 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.

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

2. $P = \lambda k.K$, then
$$\begin{array}{rcl} (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!] & \equiv & \mathcal{C}_k[\![(\mathcal{A}\ \Phi^{-1}[\![\lambda k.K]\!])]\!] \\ & = & (((\lambda k.\lambda x.x)\ k)\ (\Phi \circ \Phi^{-1})[\![\lambda k.K]\!]) \\ & \longrightarrow\!\!\!\!\!\rightarrow & (\Phi \circ \Phi^{-1})[\![\lambda k.K]\!] \end{array}$$
The result follows by in-lining 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 k_i by the procedure $(\lambda d.k_i)$ and using the inductive hypothesis, we get $(k'\ W)$.
   (b) $K = \lambda x.P'$, then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$
$$\begin{array}{rl} = & \mathcal{C}_k[\![((\lambda x.\mathcal{C}^{-1}[\![P']\!])\ \Phi^{-1}[\![W]\!])]\!] \\ = & ((\lambda x.(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P']\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]) \end{array}$$
   The result follows by the inductive hypothesis.
   (c) $K = W'K'$, then $(\mathcal{C}_k \circ \mathcal{C}^{-1})[\![P]\!]$
$$\begin{array}{rl} \equiv & (\mathcal{C}_k \circ \mathcal{C}^{-1})[\![((W'K')\ W)]\!] \\ \equiv & (((\Phi \circ \Phi^{-1})[\![W']\!]\ (\mathcal{K}_k \circ \mathcal{K}^{-1})[\![K']\!])\ (\Phi \circ \Phi^{-1})[\![W]\!]) \end{array}$$
   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 $(\Phi \circ \Phi^{-1})[\![\lambda k.K]\!]$

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

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 $\lambda k.K$.

□

The second lemma relates the terms M and $(\mathcal{C}^{-1} \circ \mathcal{C}_k)[\![M]\!]$ via the axioms AB for the pure language and the control-specific axioms C in Definition 4.4.

Lemma 4.2 Let $M \in \Lambda + callcc + \mathcal{A}$, E be an evaluation context in the same language, and k a variable that is not free in either. Then,

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

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

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

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

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

(d) $V = \mathcal{A}$, then $(\Phi^{-1} \circ \Phi)[\![V]\!] \equiv \Phi^{-1}[\![\lambda k.\lambda x.x]\!] = \lambda z.\mathit{callcc}\ \lambda k.((\lambda x.\mathcal{A}\ x)\ z)$. By a C_{elim}-reduction, we get $\lambda z.((\lambda x.\mathcal{A}\ x)\ z)$ and the result follows by a β_v-reduction and an η_v-reduction.

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

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

□

4.1.3 Equational Correspondence

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

Lemma 4.3 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:

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

Proof We consider each notion of reduction separately.

1. The reduction is: $((\lambda x.P)\ W) \longrightarrow P[x := W]$. Then we want to show that:

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

   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:

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

   (a) $P = y$, then $\mathcal{C}^{-1}[\![y]\!][x := \Phi^{-1}[\![W]\!]] \equiv (\mathcal{A}\ y) \equiv \mathcal{C}^{-1}[\![y[x := W]]\!]$.

   (b) $P = x$, then

$$\mathcal{C}^{-1}[\![x]\!][x := \Phi^{-1}[\![W]\!]] \equiv (\mathcal{A}\ \Phi^{-1}[\![W]\!]) \equiv \mathcal{C}^{-1}[\![x[x := W]]\!]$$

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

$$\begin{aligned} &\equiv (\mathcal{A}\ \Phi^{-1}[\![\lambda k.K]\!][x := \Phi^{-1}[\![W]\!]]) \\ &\equiv (\mathcal{A}\ \Phi^{-1}[\![\lambda k.K[x := W]]\!]) \qquad \text{(similar to case 7)} \\ &\equiv \mathcal{C}^{-1}[\![\lambda k.K[x := W]]\!]\end{aligned}$$

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

$$\begin{aligned}
&\equiv \mathcal{K}^{-1}[\![K']\!][x := \Phi^{-1}[\![W]\!]][\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]]] \\
&\equiv \mathcal{K}^{-1}[\![K'[x := W]]\!][\Phi^{-1}[\![W'[x := W]]\!]] \qquad \text{(induction)} \\
&\equiv \mathcal{C}^{-1}[\![(K'\ W')[x := W]]\!]
\end{aligned}$$

(e) $W' = x$, then

$$\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]] \equiv \Phi^{-1}[\![W]\!] \equiv \Phi^{-1}[\![x[x := W]]\!]$$

(f) $W' = y$, then $\Phi^{-1}[\![y]\!][x := \Phi^{-1}[\![W]\!]] \equiv y \equiv \Phi^{-1}[\![y[x := W]]\!]$.

(g) $W' = \lambda k.K$, then $\Phi^{-1}[\![W']\!][x := \Phi^{-1}[\![W]\!]]$

$$\begin{aligned}
&\equiv \lambda z.callcc\ \lambda k.\mathcal{K}^{-1}[\![K]\!][x := \Phi^{-1}[\![W]\!]][z] \\
&\equiv \lambda z.callcc\ \lambda k.\mathcal{K}^{-1}[\![K[x := W]]\!][z] \qquad \text{(induction)} \\
&\equiv \Phi^{-1}[\![\lambda k.K[x := W]]\!]
\end{aligned}$$

(h) $K = k$, then $\mathcal{K}^{-1}[\![k]\!][x := \Phi^{-1}[\![W]\!]] \equiv (k\ [\,]) \equiv \mathcal{K}^{-1}[\![k[x := W]]\!]$.

(i) $K = \lambda y.P$, 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.

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

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,

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

 - 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 sub-case applies.

3. The reduction is: $(\lambda x.K x) \longrightarrow K$ where x is not free in K. Then $\mathcal{K}^{-1}[\![\lambda x.K x]\!] \equiv ((\lambda x.\mathcal{K}^{-1}[\![K]\!][x])\ [\,])$ is equal to $\mathcal{K}^{-1}[\![K]\!]$ by β'_Ω.

4. The reduction is: $((\lambda k.K_1)\ K_2) \longrightarrow K_1[k := K_2]$. We have to show that:

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

   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:

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

   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.

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

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

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

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

   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 $D[\mathcal{K}^{-1}[\![K]\!][callcc\ \lambda k.\mathcal{K}^{-1}[\![K']\!][M]]]$

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

The main proof proceeds by case analysis:

(a) $P = x$, then:

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

(b) $P = \lambda k'.K'$. The left hand side:

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

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

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

(d) $W = x$, then the result is immediate.

(e) $W = \lambda k'.K$, then $\Phi^{-1}[\![\lambda k'.K[k := K_2]]\!]$

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

(f) $K_1 = k'$, then:

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

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

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

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

(i) $K_1 = W'K'$, then the left hand side:

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

□

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

Lemma 4.4 Let $M, N \in \Lambda + \mathit{callcc} + \mathcal{A}$, then $\boldsymbol{\lambda}\beta_v ABC \vdash M = N$ implies that $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$.

Proof Easy. □

The above lemmas imply that the calculi $\boldsymbol{\lambda}\beta_v ABC$ and $\boldsymbol{\lambda}\beta\eta$ satisfy an equational correspondence theorem.

> **Theorem 4.5** *(Equational Correspondence)* The theory $\boldsymbol{\lambda}\beta_v ABC$ equationally corresponds to the theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages $\Lambda + \mathit{callcc} + \mathcal{A}$ and $\mathit{cps}(\Lambda + \mathit{callcc} + \mathcal{A})$, and the translations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

4.1.4 Prompts

Both $\mathcal{A}$ and *callcc* allow *unrestricted* transfers of control. Denotational models for languages with such control operators naturally include elements that delimit control actions [80, 95]. In addition to their theoretical importance, these delimiters present a useful programming paradigm [93, 94]. In this section, we study one traditional control delimiter: # (prompt) [24].

Intuitively, in an expression $(\# \ M)$, the prompt treats its subexpression as a complete program by evaluating it in the initial continuation, forwarding the result to the current continuation. To specify this behavior formally, we extend the CPS transformation accordingly:

$$\mathcal{C}_k[\![(\# \ M)]\!] = (k \ \mathcal{C}_c[\![M]\!])[c := \lambda x.x]$$

Unfortunately, this extension produces CPS terms that are not tail-recursive, and hence CPS programs are no longer indifferent to the order of evaluation. For example, the expression:

$$((\lambda x.y) \ (\# \ \Omega)), \qquad \text{where } \Omega = ((\lambda x.xx) \ (\lambda x.xx))$$

translates as follows:

$$((\lambda x.ky) \ ((\lambda x.((x \ \lambda x.x) \ x)) \ (\lambda k.\lambda x.((x \ k) \ x)))).$$

This CPS expression would reduce to $(k \ y)$ in a call-by-name calculus but would diverge in a call-by-value calculus. In the source language, this choice corresponds to whether an expression of the form $(\# \ M)$ is a value or not. We refer to the prompt in the first case as the *lazy prompt* because its subexpression may not be evaluated, and refer to the other prompt as the *strict prompt*.

The classic semantics for control delimiters [21, 24] corresponds to the strict prompt. Since meta-theorems are, as always, easier to formulate and prove for the by-name version, we address the lazy prompt first and defer the strict prompt for the moment.

Lazy prompt

For the lazy prompt, the CPS language has a call-by-name semantics and therefore the reductions for the CPS language are the canonical reductions β and η. Formally, we extend the source language as follows:

$$V \ ::= \ \cdots \mid (\# \ M)$$

The CPS transformation includes one additional clause:

$$\Phi[\![(\# \ M)]\!] = \mathcal{C}_c[\![M]\!][c := \lambda x.x]$$

The set of CPS terms reflects the fact that the prompt injects answers into the domain of values. Since $\mathcal{A}$ has the opposite effect, the two syntactic categories collapse into one:

$$\begin{array}{rcll} P, W & ::= & x \mid (\lambda k.K) \mid (K \ W) & \qquad (\mathit{Answers}, \mathit{Values}) \\ K & ::= & k \mid (\lambda x.P) \mid (W \ K) & \qquad (\mathit{Continuations}) \end{array}$$

Finally, the extension to the inverse CPS transformation is straightforward:

$$\Phi^{-1}[\![(K \ W)]\!] \ = \ (\# \ \mathcal{C}^{-1}[\![(K \ W)]\!])$$

The reductions for the extended source language include all the previously defined reductions *mutatis mutandis* and the additional reductions ($C^{\#}$) of Figure 4.1. The

$$\begin{array}{rcll} (\# \ V) & \longrightarrow & V & & (\#_v) \\ (\# \ (\mathcal{A} \ M)) & \longrightarrow & (\# \ M) & & (\#_{\mathcal{A}}) \\ (\# \ (\mathit{callcc} \ M)) & \longrightarrow & (\# \ (M \ (\lambda x.(\mathcal{A} \ x)))) & x \notin FV(E) & (\#_{\mathit{callcc}}) \\ (\mathcal{A} \ (\# \ M)) & \longrightarrow & (\mathcal{A} \ M) & & (\#_{\mathit{elim}}) \end{array}$$

Figure 4.1 The Axioms $C^{\#}$ for Prompt.

new reductions reflect the essential operational properties of the prompt. The prompt disappears after its subexpression becomes a value. The prompt delimits all control actions, therefore $\mathcal{A}$ can only ignore the context up to the (dynamically) closest prompt and *callcc* can only capture the continuation up to the (dynamically) closest prompt. The last reduction shows that $\mathcal{A}$ forces the evaluation of its argument to occur next to a prompt, or equivalently, in an empty continuation.

The set of reductions is complete with respect to $\boldsymbol{\lambda}\beta\eta$ on CPS terms.

Theorem 4.6 *(Correspondence)* The theory $\boldsymbol{\lambda}\beta_v ABCC^{\#}$ equationally correspond to the theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages $\Lambda + \mathit{callcc} + \mathcal{A} + \#$ and $\mathit{cps}(\Lambda + \mathit{callcc} + \mathcal{A} + \#)$, and the transformations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

Strict prompt

The CPS transformation and its inverse remain essentially the same. However, the call-by-value semantics of the CPS language invalidates some of the reductions on CPS terms and hence some of the source reductions. The exact reductions that become invalid depend on which theory is used for reasoning about the CPS language, which is not obvious since there is no canonical theory for call-by-value languages.

Calculus for CPS terms:

$\boldsymbol{\lambda}\beta_v$

$$\begin{array}{rcl}
(\#\ V) & \longrightarrow & V \\
 & & \\
(\#\ (\mathcal{A}\ M)) & \longrightarrow & (\#\ M) \\
(\#\ (\mathit{callcc}\ M)) & \longrightarrow & (\#\ (M\ (\lambda x.(\mathcal{A}\ x))))
\end{array}$$

$\boldsymbol{\lambda}\beta_v AB$

$$\begin{array}{rcl}
(\#\ V) & \longrightarrow & V \\
(\#\ (\#\ M)) & \longrightarrow & (\#\ M) \\
(\#\ (\mathcal{A}\ M)) & \longrightarrow & (\#\ M) \\
(\#\ (\mathit{callcc}\ M)) & \longrightarrow & (\#\ (M\ (\lambda x.(\mathcal{A}\ x)))) \\
(\mathcal{A}\ (\#\ M)) & \longrightarrow & (\mathcal{A}\ M)
\end{array}$$

Figure 4.2 The Axioms $C_v^{\#}$ for Prompt.

Figure 4.2 shows the source axioms for two possible call-by-value CPS calculi: the λ_v-calculus, and the stronger $\boldsymbol{\lambda}\beta_v AB$-calculus. In both cases, expressions of the form $(\#\ M)$ are no longer values (which also affects the definition of evaluation contexts).

4.1.5 Axiomatizing Observational Equivalence

As in the case for the pure language (cf. Section 3.6.2), a correspondence between a typed version of our calculus and the simply typed λ-calculus implies a soundness and completeness theorem with respect to a natural denotational model. In this case, we can also exploit a "full abstraction" result to extend this result to a soundness and completeness theorem with respect to a notion of observational equivalence.

In the remainder of this section, we restrict the type of answers a so that it coincides with the type o of observable objects. The set of source terms is an extension of the simply typed λ-calculus with the control operators and their usual types [95]:

$$V \ ::= \ \ldots \mid callcc^{((s\to t)\to s)\to s} \mid \mathcal{A}^{o\to t} \mid (\# \ M^o)^o$$

Definition 4.5 *(CPS^t)* The full set of CPS terms is:

$$\begin{array}{rcl} P & ::= & x^a \mid (K^{t\to a} \ W^t)^a \mid (K^{a\to a} \ P^a)^a \\ W & ::= & x^t \mid (\lambda k^{t\to a}.K^{s\to a})^{(t\to a)\to s\to a} \\ K & ::= & k^{t\to a} \mid (\lambda x^t.P^a)^{t\to a} \mid (W^{(t\to a)\to s\to a} \ K^{t\to a})^{s\to a} \end{array}$$

As before, the CPS transformation and its inverse preserve the types of terms.

Lemma 4.7 Let $M \in \Lambda_c^t$, and $P \in CPS^t$, then $\mathcal{C}_k[\![M]\!] \in CPS^t$ and $\mathcal{C}^{-1}[\![P]\!] \in \Lambda_c^t$.

As a corollary, Correspondence Theorem 4.6 extends to these typed languages.[8]

Theorem 4.8 *(Correspondence for Typed Languages)* The typed theory $\boldsymbol{\lambda}\beta_v ABCC^{\#}$ equationally corresponds to the typed theory $\boldsymbol{\lambda}\beta\eta$ (relative to the languages Λ_c^t and CPS^t, and the transformations $\mathcal{C}_k$ and $\mathcal{C}^{-1}$).

The definitions of the full type structure and the CPS type structure are the straightforward generalizations of the similar definitions in Section 3.6.2. Given the completeness of the simply typed λ-calculus with respect to the full continuous type frame, it follows that the calculus $\boldsymbol{\lambda}\beta_v ABCC^{\#}$ proves all denotational equivalences in the CPS continuous type frame.

Theorem 4.9 Let $M, N \in \Lambda_c^t$, then $\boldsymbol{\lambda}\beta_v ABCC^{\#} \vdash M = N$ if and only if the CPS continuous type frame satisfies $M = N$.

[8]This theorem provides an alternative proof for Griffin's [46] weak normalization theorem for the λ_v-C-calculus. The problem of strong normalization for λ_v-C-calculus (or for our new calculus) remains open since reductions in the source language do not necessarily translate to reductions on the CPS side.

Sketch By the correspondence theorem, $\boldsymbol{\lambda}\beta\eta \vdash \mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$. Since the latter terms are also simply typed terms, the *full* continuous type frame satisfies the equation $\mathcal{C}_k[\![M]\!] = \mathcal{C}_k[\![N]\!]$ [76]. The result follows because the two denotational models define the same semantics. $\square$

Finally, the calculus $\boldsymbol{\lambda}\beta_v ABCC^{\#}$ corresponds to a restriction of the call-by-value observational equivalence that accounts for control operations.

> **Theorem 4.10** Let $M, N \in \Lambda_c^t$, then $\boldsymbol{\lambda}\beta_v ABCC^{\#} \vdash M = N$ if and only if M and N are call-by-value observationally equivalent in PCF contexts augmented with *callcc*, $\mathcal{A}$, the lazy prompt, and **pif**.

Sketch By modifying the full abstraction proof of Sitaram and Felleisen [95], we prove that the CPS continuous frame model satisfies $M = N$ if only if M and N are observationally equivalent. The rest follows from Theorem 4.9. $\square$

4.2 Assignments

Realistic programming languages include imperative, assignment-like constructs. In contrast to the previous source language extensions ($\mathcal{A}$ and *callcc*), the addition of assignments produces a CPS language that is *not* a subset of Λ. Therefore, in order to extend our correspondence theorems in such cases, we have to extend *both* the calculus for the source language *and* the calculus for the CPS language.

In this section, we address the problem of deriving a calculus for a source language with assignments. The derivation is similar to the derivation of the computational λ-calculus in the previous chapter but uses the store-passing style (SPS) transformation instead of the CPS transformation. In the next section, we derive a calculus for a CPS language augmented with assignments and extend the correspondence theorem to the richer language.

4.2.1 The Store-Passing Transformation

We consider an extension of the language Λ with simple imperative facilities. The language includes an infinite set of constants A representing locations. Locations may be created dynamically, passed as arguments, and assigned to using setref!. The contents of a location is accessed using the operation deref. The constant $()$ denotes an unspecified value:

$$
\begin{array}{rcl}
M & ::= & V \mid (M\ N) \mid (\mathsf{letref}\ A\ M) \\
V & ::= & () \mid x \mid A \mid (\lambda x.M) \mid \mathsf{deref} \mid \mathsf{setref!} \\
A & \in & \mathit{Locations} = \{a_1, a_2, \ldots\}
\end{array}
$$

Although this language is quite simple, it is sufficient to represent, *e.g.*, pointer manipulations, call-by-reference parameter passing, ML-style reference cells,[9] and dynamic allocation and de-allocation.[10]

In order to define the semantics of the source language $\Lambda + \mathit{ref}$, we exploit an idea from denotational semantics [100]. This semantics regards the store as a primitive concept, and uses a "store-passing transformation" to translate imperative operations to explicit operations on a concrete representation of the store. Typically, it is easy to reason about the resulting language by extending an appropriate λ-calculus with a store algebra:

Definition 4.6 *($sps(\Lambda + \mathit{ref})$)* The target language is:

$$
\begin{array}{rcll}
P & ::= & \langle W, S\rangle \mid (W\ P) \mid S@W \mid (\mathit{alloc}\ A\ P) & (\mathit{Terms}) \\
W & ::= & ? \mid x \mid A \mid (\lambda\langle x, s\rangle.P) & (\mathit{Values}) \\
S & ::= & s \mid (\mathit{upd}\ S\ W\ W) & (\mathit{Stores})
\end{array}
$$

The notation $\langle W, S\rangle$ denotes a pair whose first element is the value W and second element is the store S. The variable s denotes an unknown store and the term $(\mathit{upd}\ S\ W_1\ W_2)$ represents a store S whose most recent assignment associates expression W_2 with address W_1. Such a store may contain more than one assignment for any given address. The lookup operation $S@W_1$ returns a pair $\langle W_2, S\rangle$ where W_2 is the rightmost, or most recent assignment, to location W_1 in the store S. The constant

---

[9]The language can express ML-like reference cells as follows:

$$\mathsf{ref}\ =\ \lambda x.(\mathsf{letref}\ A\ ((\mathsf{setref!}\ A\ x)\ ;\ A)).$$

where the sequencing operation $M; N$ abbreviates $((\lambda d.N)\ M)$ where d does not occur in the free variables of N.

[10]Our approach to dynamic allocation has the advantage of simplicity and abstractness, but relies on the meta-operation of α-renaming to ensure uniqueness of addresses [29]. A more "concrete" allocation mechanism could be defined in any number of ways, but would require more complicated axioms to correctly reflect its internal behavior.

? denotes an unspecified value. In the term (*alloc* A P) the location A is *bound* in P. An $sps(\Lambda + \mathit{ref})$-program is a closed term $((\lambda s.P)\ \{\})$ where $\{\}$ is the "initial" empty store.

Given the above informal explanation, we present five axioms (Q) that suffice for the evaluation of $sps(\Lambda + \mathit{ref})$-programs.

Definition 4.7 *(The SPS Axioms Q)*

$$\begin{array}{rcll}
((\lambda\langle x, s\rangle.P)\ \langle W, S\rangle) & = & P[x := W, s := S] & (\beta_v^\times) \\
(\mathit{upd}\ S\ A\ W)@A & = & \langle W, (\mathit{upd}\ S\ A\ W)\rangle & (\mathit{lk}_1) \\
(\mathit{upd}\ S\ A_1\ W)@A & = & ((\lambda\langle x, s\rangle.\langle x, (\mathit{upd}\ s\ A_1\ W)\rangle)\ S@A) & (\mathit{lk}_2) \\
 & & A_1 \neq A & \\
(W\ (\mathit{alloc}\ A\ P)) & = & (\mathit{alloc}\ A\ (W\ P)) \qquad A \notin FV(W) & (l_{\mathit{lift}}) \\
(\mathit{alloc}\ A\ P) & = & P \qquad A \notin FV(P) & (\mathit{gc})
\end{array}$$

Axioms $\beta_v^\times$ and lk_1 are straightforward and self-explanatory. Axiom lk_2 ensures that store components *not* containing the looked-up address are preserved. Axiom l_{lift} implements scope extrusion [29, 66]; it "lifts" the binding of the location A to permit the evaluation of the application $(W\ P)$. The condition $A \notin FV(W)$ ensures that *alloc* does not capture any free locations A in W and may require that the bound location A be renamed, *e.g.*,

$$((\lambda\langle d, s\rangle.s@A)\ (\mathit{alloc}\ A\ \langle A, S\rangle)) = (\mathit{alloc}\ A_1\ ((\lambda\langle d, s\rangle.s@A)\ \langle A_1, S\rangle)).$$

The last axiom is a garbage collection rule for de-allocating references.

The semantics of the source language $(\Lambda + \mathit{ref})$ is defined indirectly by the axioms Q, using the following store-passing translation.

Definition 4.8 *(Store-Passing Transformation)*

$$\begin{array}{rcl}
\mathcal{S} : (\Lambda + \mathit{ref}) \times S & \rightarrow & sps(\Lambda + \mathit{ref}) \\
\mathcal{S}[\![V]\!]s & = & \langle \mathcal{W}[\![V]\!], s\rangle \\
\mathcal{S}[\![(M\ N)]\!]s & = & ((\lambda\langle x, s\rangle.(x\ \mathcal{S}[\![N]\!]s))\ \mathcal{S}[\![M]\!]s) \\
\mathcal{S}[\![(\mathsf{letref}\ A\ M)]\!]s & = & (\mathit{alloc}\ A\ \mathcal{S}[\![M]\!]s)
\end{array}$$

$$
\begin{aligned}
\mathcal{W} : (\Lambda + \mathit{ref})(V) &\rightarrow \mathit{sps}(\Lambda + \mathit{ref})(W) \\
\mathcal{W}[\![()]\!] &= ? \\
\mathcal{W}[\![x]\!] &= x \\
\mathcal{W}[\![A]\!] &= A \\
\mathcal{W}[\![(\lambda x.M)]\!] &= (\lambda\langle x, s\rangle.\mathcal{S}[\![M]\!]s) \\
\mathcal{W}[\![\mathsf{deref}]\!] &= (\lambda\langle x, s\rangle.s@x) \\
\mathcal{W}[\![\mathsf{setref!}]\!] &= (\lambda\langle x, s\rangle.\langle(\lambda\langle y, s'\rangle.\langle ?, (\mathit{upd}\ s'\ x\ y)\rangle), s\rangle)
\end{aligned}
$$

4.2.2 Inverting the Store-Passing Transformation

The specification of an inverse to the store-passing translation relies on the representation of stores in the source language ($\Lambda + \mathit{ref}$). We choose the following common representation [8, 62, 73]. A store of the form:

$$(\mathit{upd}\ (\mathit{upd}\ \ldots\ (\mathit{upd}\ (\mathit{upd}\ s\ W_{11}\ W_{12})\ W_{21}\ W_{22})\ \ldots)\ W_{n1}\ W_{n2})$$

will be represented in the source language as the sequence of assignments:

$$()\ ;\ (\mathsf{setref!}\ V_{11}\ V_{12})\ ;\ (\mathsf{setref!}\ V_{21}\ V_{22})\ ;\ \ldots\ ;\ (\mathsf{setref!}\ V_{n1}\ V_{n2}),$$

where V_{ij} corresponds to W_{ij}, the sequencing operation $(M\ ;\ N)$ is an abbreviation for: $((\lambda d.N)\ M)$, where d is not free in N.

The following inverse store-passing transformation defines inverses for arbitrary terms of $\mathit{sps}(\Lambda + \mathit{ref})$, and formalizes the store's representation.

Definition 4.9 *(Inverse Store-Passing)*

$$
\begin{aligned}
\mathcal{S}^{-1} : \mathit{sps}(\Lambda + \mathit{ref}) &\rightarrow (\Lambda + \mathit{ref}) \\
\mathcal{S}^{-1}[\![\langle W, S\rangle]\!] &= \mathcal{I}[\![S]\!]\ ;\ \mathcal{W}^{-1}[\![W]\!] \\
\mathcal{S}^{-1}[\![(W\ P)]\!] &= (\mathcal{W}^{-1}[\![W]\!]\ \mathcal{S}^{-1}[\![P]\!]) \\
\mathcal{S}^{-1}[\![S@W]\!] &= \mathcal{I}[\![S]\!]\ ;\ (\mathsf{deref}\ \mathcal{W}^{-1}[\![W]\!]) \\
\mathcal{S}^{-1}[\![(\mathit{alloc}\ A\ P)]\!] &= (\mathsf{letref}\ A\ \mathcal{S}^{-1}[\![P]\!])
\end{aligned}
$$

$$\begin{array}{rcl}
\mathcal{W}^{-1}[\![?]\!] & = & () \\
\mathcal{W}^{-1}[\![x]\!] & = & x \\
\mathcal{W}^{-1}[\![A]\!] & = & A \\
\mathcal{W}^{-1}[\![(\lambda\langle x, s\rangle.P)]\!] & = & (\lambda x.\mathcal{S}^{-1}[\![P]\!])
\end{array}$$

$$\begin{array}{rcl}
\mathcal{I}[\![s]\!] & = & () \\
\mathcal{I}[\![(\mathit{upd}\ S\ W_1\ W_2)]\!] & = & \mathcal{I}[\![S]\!]\ ;\ (\mathsf{setref!}\ \mathcal{W}^{-1}[\![W_1]\!]\ \mathcal{W}^{-1}[\![W_2]\!])
\end{array}$$

A translation to store-passing style immediately followed by an inverse translation produces a term that is related to the original via the following axioms.

Definition 4.10 *(Administrative Axioms: $D = \{\eta_v, \beta'_\Omega, D_1, D_2\}$)*

$$\begin{array}{rcll}
((\lambda x.M)\ V) & = & M[x := V] & (\beta_v) \\
\mathsf{deref} & = & (\lambda x.(\mathsf{deref}\ x)) & (\eta_v) \\
\mathsf{setref!} & = & (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) & (\eta_v) \\
(V\ ((\mathsf{setref!}\ V_1\ V_2)\ ;\ M)) & = & (\mathsf{setref!}\ V_1\ V_2)\ ;\ (V\ M) & (D_1) \\
(\mathsf{letref}\ A\ ((\mathsf{setref}\ V_1\ V_2)\ ;\ M)) & = & (\mathsf{setref!}\ V_1\ V_2);\ (\mathsf{letref}\ A\ M) & (D_2) \\
 & & A \notin FV(V_1, V_2) & \\
(M\ N) & = & ((\lambda x.(x\ N))\ M) & (\beta'_\Omega)
\end{array}$$

We now prove a tight relationship between the original store-passing transformation and its inverse.

Lemma 4.11 Let $M \in (\Lambda + \mathit{ref})$ and S be a store in $\mathit{sps}(\Lambda + \mathit{ref})$. Then

$$\boldsymbol{\lambda}\beta_v D \vdash (\mathcal{I}[\![S]\!]\ ;\ M) = \mathcal{S}^{-1}[\![\mathcal{S}[\![M]\!]S]\!]$$

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

- $M = V$, then $\mathcal{S}^{-1}[\![\mathcal{S}[\![V]\!]S]\!] = \mathcal{S}^{-1}[\![\langle\mathcal{W}[\![V]\!], S\rangle]\!] = (\mathcal{I}[\![S]\!]\ ;\ \mathcal{W}^{-1}[\![\mathcal{W}[\![V]\!]]\!])$. It remains to show that:

$$\boldsymbol{\lambda}\beta_v D \vdash \mathcal{W}^{-1}[\![\mathcal{W}[\![V]\!]]\!] = V.$$

  - $V = x$ or $V = A$, then the result is immediate.

– $V = \lambda x.N$, then:

$$\begin{aligned}
\mathcal{W}^{-1}[\![\mathcal{W}[\![(\lambda x.N)]\!]]\!] &= \mathcal{W}^{-1}[\![(\lambda\langle x,s\rangle.\mathcal{S}[\![N]\!]s)]\!] \\
&= (\lambda x.\mathcal{S}^{-1}[\![\mathcal{S}[\![N]\!]s]\!]) \\
&= (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ N)) && \text{(induction)} \\
&= (\lambda x.(()\ ;\ N)) \\
&= (\lambda x.N) && (\beta_v)
\end{aligned}$$

– $V = \textsf{deref}$, then:

$$\begin{aligned}
\mathcal{W}^{-1}[\![\mathcal{W}[\![\textsf{deref}]\!]]\!] &= \mathcal{W}^{-1}[\![(\lambda\langle x,s\rangle.s@x)]\!] \\
&= (\lambda x.\mathcal{S}^{-1}[\![s@x]\!]) \\
&= (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ (\textsf{deref}\ x)))
\end{aligned}$$

– $V = \textsf{setref!}$, then:

$$\begin{aligned}
\mathcal{W}^{-1}[\![\mathcal{W}[\![\textsf{setref!}]\!]]\!] &= \mathcal{W}^{-1}[\![(\lambda\langle x,s\rangle.\langle(\lambda\langle y,s'\rangle.\langle ?,(\mathit{upd}\ s'\ x\ y)\rangle),s\rangle)]\!] \\
&= (\lambda x.(\mathcal{I}[\![s]\!]\ ;\ \lambda y.(\mathcal{I}[\![s']\!]\ ;\ (\textsf{setref!}\ x\ y)\ ;\ ?)))
\end{aligned}$$

- $M = (N_1\ N_2)$, then:

$$\begin{aligned}
\mathcal{S}^{-1}[\![\mathcal{S}[\![(N_1\ N_2)]\!]S]\!] &= \mathcal{S}^{-1}[\![((\lambda\langle x,s\rangle.(x\ \mathcal{S}[\![N_2]\!]s))\ \mathcal{S}[\![N_1]\!]S)]\!] \\
&= ((\lambda x.(x\ (\mathcal{I}[\![s]\!]\ ;\ N_2)))\ (\mathcal{I}[\![S]\!]\ ;\ N_1)) \\
&= ((\lambda x.(x\ N_2))\ (\mathcal{I}[\![S]\!]\ ;\ N_1)) && (\beta_v) \\
&= \mathcal{I}[\![S]\!]\ ;\ ((\lambda x.(x\ N_2))\ N_1) \\
&\quad\ \text{(Repeated application of } (D_1)) \\
&= \mathcal{I}[\![S]\!]\ ;\ (N_1\ N_2) && (\beta'_\Omega)
\end{aligned}$$

- $M = (\textsf{letref}\ A\ N)$, then:

$$\begin{aligned}
\mathcal{S}^{-1}[\![\mathcal{S}[\![(\textsf{letref}\ A\ N)]\!]S]\!] &= \mathcal{S}^{-1}[\![(\mathit{alloc}\ A\ \mathcal{S}[\![N]\!]S)]\!] \\
&= (\textsf{letref}\ A\ \mathcal{S}^{-1}[\![\mathcal{S}[\![N]\!]S]\!]) \\
&= (\textsf{letref}\ A\ (\mathcal{I}[\![S]\!]\ ;\ N)) \\
&= \mathcal{I}[\![S]\!]\ ;\ (\textsf{letref}\ A\ N) \\
&\quad\ \text{(Repeated application of } (D_2))
\end{aligned}$$

$\square$

4.2.3 Inverting the Store-Passing Axioms

To derive the complete equational theory that corresponds to the SPS axioms Q, we apply the inverse store-passing transformation to each of these axioms. The result of inverting lk_1, lk_2, l_{lift} and gc yields the following source axioms.

Definition 4.11 *(Source axioms: $E = \{E_1, E_2, E_3, E_4\}$)*

$$
\begin{array}{rcll}
(\mathsf{setref!}\ A\ V);(\mathsf{deref}\ A) & = & (\mathsf{setref!}\ A\ V)\ ;\ V & (E_1)\\
(\mathsf{setref!}\ A_1\ V);(\mathsf{deref}\ A) & = & ((\lambda x.((\mathsf{setref!}\ A_1\ V);x))(\mathsf{deref}\ A)) & (E_2)\\
 & & A_1 \neq A & \\
(\mathsf{letref}\ A\ (V\ M)) & = & (V\ (\mathsf{letref}\ A\ M)) \qquad A \notin FV(V) & (E_3)\\
(\mathsf{letref}\ A\ M) & = & M \qquad A \notin FV(M) & (E_4)
\end{array}
$$

The remaining axiom, $\beta_v^\times$, cannot be inverted directly since it involves substitution. However, it turns out that the equivalence between the inversions of its right side and its left side can be established using the set of axioms we have already developed. To show this requires a number of lemmas.

Lemma 4.12 *(Substitution)* Let P and W be terms of the appropriate sort in $sps(\Lambda + ref)$, then $\mathcal{S}^{-1}[\![P]\!][x := \mathcal{W}^{-1}[\![V]\!]]$ is identical to $\mathcal{S}^{-1}[\![P[x := W]]\!]$.

Proof Idea The proof is by induction on the structure of P and proceeds by cases. □

Lemma 4.13 Let P, S and S_1 be terms of the appropriate sort in $sps(\Lambda + ref)$, then:

- $\boldsymbol{\lambda}\beta_v DE \vdash (\mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!]) = \mathcal{S}^{-1}[\![P[s := S]]\!]$.
- $\boldsymbol{\lambda}\beta_v DE \vdash (\mathcal{I}[\![S]\!]\ ;\ \mathcal{I}[\![S_1]\!]) = \mathcal{I}[\![S_1[s := S]]\!]$.

Proof The proof is by induction on the structure of P or S_1. The proof relies on the fact that there is exactly one free store variable s in terms P and S and no free store variables in W:

- $P = \langle W_1, S_1 \rangle$, then:

$$\begin{aligned} \mathcal{I}[\![S]\!] \ ; \ \mathcal{S}^{-1}[\![P]\!] &= \mathcal{I}[\![S]\!] \ ; \ \mathcal{I}[\![S_1]\!] \ ; \ \mathcal{W}^{-1}[\![W_1]\!] \\ &= \mathcal{I}[\![S_1[s := S]]\!] \ ; \ \mathcal{W}^{-1}[\![W_1]\!] \\ &= \mathcal{S}^{-1}[\![\langle W_1, S_1[s := S] \rangle]\!] \end{aligned}$$

- $P = (W_1 \ P_1)$, then:

$$\begin{aligned} \mathcal{I}[\![S]\!] \ ; \ \mathcal{S}^{-1}[\![P]\!] &= \mathcal{I}[\![S]\!] \ ; \ (\mathcal{W}^{-1}[\![W_1]\!] \ \mathcal{S}^{-1}[\![P_1]\!]) \\ &= (\mathcal{W}^{-1}[\![W_1]\!] \ (\mathcal{I}[\![S]\!] \ ; \ \mathcal{S}^{-1}[\![P_1]\!])) \\ &= (\mathcal{W}^{-1}[\![W_1]\!] \ \mathcal{S}^{-1}[\![P_1[s := S_1]]\!]) \end{aligned}$$

- $P = S_1 @ W_1$, then:

$$\begin{aligned} \mathcal{I}[\![S]\!] \ ; \ \mathcal{S}^{-1}[\![P]\!] &= \mathcal{I}[\![S]\!] \ ; \ \mathcal{I}[\![S_1]\!] \ ; \ (\mathsf{deref} \ \mathcal{W}^{-1}[\![W_1]\!]) \\ &= \mathcal{I}[\![S_1[s := S]]\!] \ ; \ (\mathsf{deref} \ \mathcal{W}^{-1}[\![W_1]\!]) \\ &= \mathcal{S}^{-1}[\![S_1[s := S] @ W_1]\!] \end{aligned}$$

- $S_1 = s$, then:

$$\begin{aligned} \mathcal{I}[\![S]\!] \ ; \ \mathcal{I}[\![S_1]\!] &= () \ ; \ \mathcal{I}[\![S_1]\!] \\ &= \mathcal{I}[\![S_1]\!] \\ &= \mathcal{I}[\![S_1[s := s]]\!] \end{aligned}$$

- $S_1 = (upd \ S_2 \ W_1 \ W_2)$, then:

$$\begin{aligned} \mathcal{I}[\![S]\!] \ ; \ \mathcal{I}[\![S_1]\!] &= \mathcal{I}[\![S]\!] \ ; \ \mathcal{I}[\![S_2]\!] \ ; \ (\mathsf{setref!} \ \mathcal{W}^{-1}[\![W_1]\!] \ \mathcal{W}^{-1}[\![W_2]\!]) \\ &= \mathcal{I}[\![S_2[s := S]]\!] \ ; \ (\mathsf{setref!} \ \mathcal{W}^{-1}[\![W_1]\!] \ \mathcal{W}^{-1}[\![W_2]\!]) \\ &= \mathcal{I}[\![(upd \ S_2[s := S] \ W_1 \ W_2)]\!] \end{aligned}$$

$\square$

Using Lemmas 4.12 and 4.13, we are now in a position to show that axiom $\beta_v^\times$ is invertible.

Lemma 4.14 Let P, S, and W be terms of the appropriate sort in $sps(\Lambda + ref)$. Then

$$\boldsymbol{\lambda}\beta_v DE \vdash \mathcal{S}^{-1}[\![((\lambda\langle x, s \rangle.P) \ \langle W, S \rangle)]\!] = \mathcal{S}^{-1}[\![P[x := W, s := S]]\!]$$

Proof The inverse of the left-hand side of the equation is:

$$\begin{aligned}
((\lambda x.\mathcal{S}^{-1}[\![P]\!])\,(\mathcal{I}[\![S]\!]\ ;\ \mathcal{W}^{-1}[\![W]\!])) &= \mathcal{I}[\![S]\!]\ ;\ ((\lambda x.\mathcal{S}^{-1}[\![P]\!])\,\mathcal{W}^{-1}[\![W]\!]) \\
&\quad \text{(Repeated application of } (D_1)) \\
&= \mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P]\!][x := \mathcal{W}^{-1}[\![W]\!]] \qquad (\beta_v) \\
&= \mathcal{I}[\![S]\!]\ ;\ \mathcal{S}^{-1}[\![P[x := W]]\!] \\
&\quad \text{(Substitution Lemma 4.12)} \\
&= \mathcal{S}^{-1}[\![P[x := W, s := S]]\!] \\
&\quad \text{(Lemma 4.13)}
\end{aligned}$$

□

The complete set of source axioms in Figure 4.3 consists of all the previously derived axioms. It proves the same equations that the axioms Q prove on store-passing terms:

$$\begin{array}{rcll}
((\lambda x.M)\ V) &=& M[x := V] & (\beta_v) \\
\mathsf{deref} &=& (\lambda x.(\mathsf{deref}\ x)) & (\eta_v) \\
\mathsf{setref!} &=& (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) & (\eta_v) \\
(V\ ((\mathsf{setref!}\ V_1\ V_2)\ ;\ M)) &=& (\mathsf{setref!}\ V_1\ V_2)\ ;\ (V\ M) & (D_1) \\
(\mathsf{letref}\ A\ ((\mathsf{setref}\ V_1\ V_2)\ ;\ M)) &=& (\mathsf{setref!}\ V_1\ V_2)\ ;\ (\mathsf{letref}\ A\ M) & (D_2) \\
 && A \notin FV(V_1, V_2) & \\
(M\ N) &=& ((\lambda x.(x\ N))\ M) & (\beta'_\Omega) \\
(\mathsf{setref!}\ A\ V)\ ;\ (\mathsf{deref}\ A) &=& (\mathsf{setref!}\ A\ V)\ ;\ V & (E_1) \\
(\mathsf{setref!}\ A_1\ V)\ ;\ (\mathsf{deref}\ A) &=& ((\lambda x.((\mathsf{setref!}\ A_1\ V)\ ;\ x))\ (\mathsf{deref}\ A)) & (E_2) \\
 && A_1 \neq A & \\
(\mathsf{letref}\ A\ (V\ M)) &=& (V\ (\mathsf{letref}\ A\ M)) \qquad A \notin FV(V) & (E_3) \\
(\mathsf{letref}\ A\ M) &=& M \qquad A \notin FV(M) & (E_4)
\end{array}$$

Figure 4.3 The Axioms DE for Reference Cells.

Theorem 4.15 $\boldsymbol{\lambda}\beta_v DE \vdash M = N$ if and only if $Q \vdash \mathcal{S}[\![M]\!]s = \mathcal{S}[\![N]\!]s$ (where s is a free store variable).

Proof Each direction is by induction on the length of proofs, using Lemmas 4.11 and 4.14, and the fact that $\mathcal{S}^{-1}$ is compositional. □

4.2.4 Assessment

To understand the behavior of imperative programs, we typically have to deal with both the inherent behavior of the store, and the complications of sequencing operations on the store amidst other fundamental operations. At some intuitive level, our derived logic is more complex than the store-augmented lambda-calculus, reflecting the intricacies of modeling the sequencing of imperative operations. When the store *sequencing* aspects of the language are encoded by translation to a target calculus, our approach allows source axioms to be developed that mirror the behavior of the translation, independently of the axioms that merely "implement the store."

Although our derived source axioms are complete *relative* to the axioms of the store algebra, the latter theory does not provide a complete axiomatization of the observational equivalence relation, and is therefore *semantically* incomplete. Obviously, we cannot achieve semantic completeness with an untyped language such as the one used in the previous section. However, the incompleteness problem is more fundamental. Even in a simply typed language, assignments can be used to define recursive functions or cyclic data structures, *e.g.*, the evaluation of the following simply typed program diverges:[11]

$$(\mathsf{setref!}\ A^{ref(int \to int)}\ (\lambda x^{int}.x^{int}))\ ;\ (\mathsf{setref!}\ A\ (\lambda x.((\mathsf{deref}\ A)\ x)))\ ;\ ((\mathsf{deref}\ A)\ 0).$$

Because of the inherent semantic incompleteness, we can always derive more powerful axiomatic systems. For example, we can extend the axioms for the imperative fragments to include equations that axiomatize the store data-type. This is the approach taken in the logic Pim [32], and we consider a few similar extensions here. For instance, we may include the following axioms:

$$\begin{aligned}
(upd\ (upd\ S\ A_1\ W_1)\ A_2\ W_2) &= (upd\ (upd\ S\ A_2\ W_2)\ A_1\ W_1) \\
&\quad A_1 \neq A_2 \text{ and } A_1 \notin FV(W_2) \\
(upd\ (upd\ S\ A\ W_1)\ A\ W_2) &= (upd\ S\ A\ W_2)
\end{aligned}$$

[11] It is possible to use a type system that rejects the above program and that guarantees that the evaluation of "simply typed" programs terminates [102]. With such restrictions, it may be possible to axiomatize the observational equivalence relation.

The corresponding axioms in the source language are immediately obtained by applying the inverse store-passing transformation:

$$
\begin{aligned}
(\mathsf{setref!}\ A_1\ V_1)\ ;\ (\mathsf{setref!}\ A_2\ V_2) &= (\mathsf{setref!}\ A_2\ V_2)\ ;\ (\mathsf{setref!}\ A_1\ V_1) \\
&\ A_1 \neq A_2 \text{ and } A_1 \notin FV(V_2) \\
(\mathsf{setref!}\ A\ V_1)\ ;\ (\mathsf{setref!}\ A\ V_2) &= (\mathsf{setref!}\ A\ V_2)
\end{aligned}
$$

We can also extend the axioms for the functional fragments to a more powerful call-by-value logic such as the computational λ-calculus. Because the sequencing operation, *i.e.*, **begin**, can be expressed using procedures, extensions to the functional fragment prove more equations between imperative terms.

4.3 Core Scheme *CS*

We are now ready to establish the equational correspondence theorem for a small, but realistic language: Core Scheme (CS). Our language extends the pure language Λ with control operations, assignments, basic and functional constants, and conditionals. The set of basic constants includes numerals and boolean constants; 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 4.1.1:

$$
\begin{array}{rcl}
M & ::= & \ldots \mid E[(\mathbf{if}\ V\ M\ M)] \mid E[(O\ V)] \mid E[(O\ V\ V)] \mid E[(\mathsf{letref}\ A\ M)] \\
V & ::= & \ldots \mid () \mid A \mid c \\
E & ::= & \ldots \mid (\mathbf{if}\ E\ M\ M) \mid (O\ E) \mid (O\ E\ M) \mid (O\ V\ E) \\
c & ::= & \mathsf{true} \mid \mathsf{false} \mid \ulcorner n \urcorner \\
O & ::= & \mathsf{integer?} \mid \mathsf{add1} \mid / \mid \mathsf{deref} \mid \mathsf{setref!} \\
A & \in & \mathit{Locations} = \{a_1, a_2, \ldots\} \\
n & \in & \mathbb{N}
\end{array}
$$

Informally, integer? recognizes integers, add1 denotes the increment function, / is the integer division operator. The symbols (), A, letref, deref, and setref! have the same explanation as in the last section.

The formal semantics of the new constructs is specified by the set of axioms F in Figure 4.4. The first three sets of axioms are straightforward; the last set specifies the semantics of reference cells as explained in the previous section. (In the presence

The Axioms F:

$$\begin{aligned}
(\textsf{if true}\ M\ N) &= M && (If_t)\\
(\textsf{if false}\ M\ N) &= N && (If_f)\\
\\
(\textsf{add1}\ \ulcorner n \urcorner) &= \ulcorner n+1 \urcorner && (Add)\\
(/\ \ulcorner n \urcorner\ \ulcorner m \urcorner) &= \ulcorner n/m \urcorner \qquad m \neq 0 && (Div)\\
(/\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) &= (\mathcal{A}\ ()) && (Div_e)\\
\\
(\textsf{integer?}\ \ulcorner n \urcorner) &= \textsf{true} && (Int_t)\\
(\textsf{integer?}\ V) &= \textsf{false} \qquad V \neq \ulcorner n \urcorner && (Int_f)
\end{aligned}$$

The Axioms DE:

$$\begin{aligned}
(\textsf{letref}\ A\ ((\textsf{setref}\ V_1\ V_2)\ ;\ M)) &= (\textsf{setref!}\ V_1\ V_2)\ ;\ (\textsf{letref}\ A\ M) && (D_2)\\
&\ A \notin FV(V_1, V_2)\\
(\textsf{setref!}\ A\ V)\ ;\ (\textsf{deref}\ A) &= (\textsf{setref!}\ A\ V)\ ;\ V && (E_1)\\
(\textsf{setref!}\ A_1\ V)\ ;\ (\textsf{deref}\ A) &= ((\lambda x.((\textsf{setref!}\ A_1\ V)\ ;\ x))\ (\textsf{deref}\ A)) && (E_2)\\
&\ A_1 \neq A\\
(\textsf{letref}\ A\ (V\ M)) &= (V\ (\textsf{letref}\ A\ M)) \qquad A \notin FV(V) && (E_3)\\
(\textsf{letref}\ A\ M) &= M \qquad A \notin FV(M) && (E_4)
\end{aligned}$$

Figure 4.4 The Axioms DEF for Core Scheme.

of β_{lift}, β_Ω, some of the axioms derived in the last section are redundant and have been omitted from the figure.)

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

$$\begin{aligned}
M &= (\textsf{if true}\ M\ N) && (If_t)\\
&= (\textsf{if}\ (\textsf{integer?}\ \ulcorner 0 \urcorner)\ M\ N) && (Int_t)\\
&= (\textsf{if}\ (\textsf{integer?}\ (\lambda x.(\ulcorner 0 \urcorner\ x)))\ M\ N) && (\eta_v)\\
&= (\textsf{if false}\ M\ N) && (Int_f)\\
&= N && (If_f)
\end{aligned}$$

To avoid the consistency problem, we restrict the equational theory for the source language by eliminating η_v. In the next section, we will add a sound and restricted version of η_v that applies only to functional constants [45].

4.3.1 The CPS Language: $cps(CS)$

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 4.2 with the following additional clauses:

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

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 4.5 specifies this semantics precisely.

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 2.1), 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 the value $()$ under call-by-value and yields $\ulcorner 8 \urcorner$ under call-by-name.

4.3.2 Equational Correspondence

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

$$
\begin{array}{rcll}
(\textsf{if true}\ P\ Q) & = & P & (\mathit{Ifk}_t) \\
(\textsf{if false}\ P\ Q) & = & Q & (\mathit{Ifk}_f) \\
\\
(\textsf{add1}_k\ K\ \ulcorner n \urcorner) & = & (K\ \ulcorner n+1 \urcorner) & (\mathit{Addk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & (K\ \ulcorner n/m \urcorner) \qquad m \neq 0 & (\mathit{Divk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & () & (\mathit{Divk}_e) \\
\\
(\textsf{integer?}_k\ K\ \ulcorner n \urcorner) & = & (K\ \textsf{true}) & (\mathit{Intk}_t) \\
(\textsf{integer?}_k\ K\ W) & = & (K\ \textsf{false}) \quad W \neq \ulcorner n \urcorner & (\mathit{Intf}_t) \\
\\
(\textsf{letref}\ A\ (\textsf{setref!}_k\ (\lambda d.P)\ W_1\ W_2)) & = & (\textsf{setref!}_k\ (\lambda d.(\textsf{letref}\ A\ P))\ W_1\ W_2) & (\mathit{Dk}_2) \\
 & & A \notin FV(W_1, W_2), d \notin FV(P) & \\
(\textsf{setref!}_k\ (\lambda d.(\textsf{deref}_k\ K\ A))\ A\ W) & = & (\textsf{setref!}_k\ (\lambda d.KW)\ A\ W) \qquad d \notin FV(K) & (\mathit{Ek}_1) \\
(\textsf{setref!}_k\ (\lambda d.(\textsf{deref}_k\ K\ A))\ A_1\ W) & = & (\textsf{deref}_k\ (\lambda x.(\textsf{setref!}_k\ (\lambda d.Kx)\ A_1\ W))\ A) & (\mathit{Ek}_2) \\
 & & A_1 \neq A, d \notin FV(P) & \\
(\textsf{letref}\ A\ P) & = & P \qquad A \notin FV(P) & (\mathit{Ek}_4)
\end{array}
$$

Figure 4.5 The Axioms $D_k E_k F_k$ for CPS Core Scheme.

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

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

$$
\begin{array}{rcl}
\Phi[\![()]\!] & = & () \\
\Phi[\![A]\!] & = & A \\
\Phi[\![c]\!] & = & c
\end{array}
$$

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

The clause for conditionals duplicates the evaluation context E in both branches of the conditional expression. This duplication is due to two factors:[12]

1. First, the Fischer CPS transformation for conditionals duplicates the continuation variable k:

$$\mathcal{F}[\![(\textbf{if}\ M\ N\ L)]\!] = \lambda k.\mathcal{F}[\![M]\!]\ (\lambda u.(\textbf{if}\ u\ \mathcal{F}[\![N]\!]k\ \mathcal{F}[\![L]\!]k)).$$

2. Second, the transformation $\mathcal{C}_k$ eliminates all the administrative redexes from the output of the Fischer CPS transformation, which instantiates the evaluation context around the conditional once for each occurrence of k.

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

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

$$\Phi^{-1}[\![c]\!] = c$$

In order to establish the correspondence between the source and CPS calculi, we need to prove results similar to Lemmas 4.1, 4.2, 4.3, and 4.4. 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 4.1 does not rely on η_w.

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

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

The first translation is used by compilers [97] but also duplicates the entire evaluation context once we close the language under $\beta\eta$-reductions. The second translation relies on Allison's [2] CPS translation that keeps the local transfer of control independent of the continuation.

- The proof of Lemma 4.2 only uses a restricted version of η_v on continuations, $\mathcal{A}$, and *callcc*:

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

  The proof also requires the introduction of the following axioms:

$$\begin{aligned} E[(\textbf{if}\ M\ N\ L)] &= (\textbf{if}\ M\ E[N]\ E[L]) && (\mathit{If}_{\mathit{lift}}) \\ \mathsf{deref} &= (\lambda x.(\mathsf{deref}\ x)) && (\eta_{v4}) \\ \mathsf{setref!} &= (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) && (\eta_{v5}) \end{aligned}$$

  The first axiom is introduced by the compacting phase of the CPS transformation and is thus an administrative call-by-value reduction (cf. Definition 3.1).

- The proof of Lemma 4.3 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 4.4 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 $\beta\eta D_k E_k F_k$ excluding η_w.

4.3.3 Summary of the Results

The full equational theories for CS and $cps(CS)$ are in Figures 4.6, 4.7, and 4.8. The correspondences between the different axiom systems for the sub-languages in the previous sections are summarized in the following table.

Language	Call-by-Value Theory	CPS Theory	η_v/η_w	Typed
Λ	AB^-	$\beta\eta_k$	$\surd$	$\surd$
$\Lambda + \mathit{callcc} + \mathcal{A}$	AB^-C'	$\beta\eta_k$	$\surd$	$\surd$
CS	$AB^-C'DEF^+$	$\beta\eta_k D_k E_k F_k$		$\surd$

The Axioms AB^-:

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

The Axioms C':

$$\begin{array}{rcll}
E[(\mathit{callcc}\ M)] & = & \mathit{callcc}\ \lambda k.E[(M\ (\lambda f.(k\ E[f])))] & (C_{\mathit{lift}}) \\
 & & k, f \notin FV(E, M) & \\
(\mathit{callcc}\ \lambda k.((\lambda x.M)\ N)) & = & ((\lambda x.(\mathit{callcc}\ \lambda k.M))\ N) & (C_{\mathit{tail}}) \\
 & & k \notin FV(N) & \\
(\mathit{callcc}\ \lambda k.kM) & = & (\mathit{callcc}\ \lambda k.M) & (C_{\mathit{cur}}) \\
(\mathit{callcc}\ \lambda d.M) & = & M \qquad d \notin FV(M) & (C_{\mathit{elim}}) \\
\mathit{callcc}\ \lambda k.C[E[(k\ M)]] & = & \mathit{callcc}\ \lambda k.C[(k\ M)] & (C_{\mathit{abort}}) \\
 & & k \notin \mathit{trap}(C) & \\
E[(\mathcal{A}\ M)] & = & (\mathcal{A}\ M) & (\mathit{Abort}) \\
(\mathit{callcc}\ (\lambda k.C[(\lambda x.kx)])) & = & (\mathit{callcc}\ (\lambda k.C[k])) & (\eta_{v1}) \\
(\lambda x.\mathit{callcc}\ \lambda k.xk) & = & \mathit{callcc} & (\eta_{v2}) \\
(\lambda x.\mathcal{A}x) & = & \mathcal{A} & (\eta_{v3})
\end{array}$$

Figure 4.6 The Theory λCS for Core Scheme (I).

The Axioms DEF^{+}:

$$
\begin{array}{rcll}
(\textbf{if}\ \mathsf{true}\ M\ N) & = & M & (\mathit{If}_t) \\
(\textbf{if}\ \mathsf{false}\ M\ N) & = & N & (\mathit{If}_f) \\
E[(\textbf{if}\ M\ N\ L)] & = & (\textbf{if}\ M\ E[N]\ E[L]) & (\mathit{If}_{\mathit{lift}}) \\
(\mathsf{add1}\ \ulcorner n \urcorner) & = & \ulcorner n+1 \urcorner & (\mathit{Add}) \\
(/\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & \ulcorner n/m \urcorner \qquad m \neq 0 & (\mathit{Div}) \\
(/\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & \ulcorner 5 \urcorner & (\mathit{Div}_e) \\
(\mathsf{integer?}\ \ulcorner n \urcorner) & = & \mathsf{true} & (\mathit{Int}_t) \\
(\mathsf{integer?}\ V) & = & \mathsf{false} \qquad V \neq \ulcorner n \urcorner & (\mathit{Int}_f) \\
(\mathsf{letref}\ A\ ((\mathsf{setref}\ V_1\ V_2)\ ;\ M)) & = & (\mathsf{setref!}\ V_1\ V_2)\ ;\ (\mathsf{letref}\ A\ M) & (D_2) \\
 & & A \notin FV(V_1, V_2) & \\
(\mathsf{setref!}\ A\ V)\ ;\ (\mathsf{deref}\ A) & = & (\mathsf{setref!}\ A\ V)\ ;\ V & (E_1) \\
(\mathsf{setref!}\ A_1\ V)\ ;\ (\mathsf{deref}\ A) & = & ((\lambda x.((\mathsf{setref!}\ A_1\ V)\ ;\ x))\ (\mathsf{deref}\ A)) & (E_2) \\
 & & A_1 \neq A & \\
(\mathsf{letref}\ A\ (V\ M)) & = & (V\ (\mathsf{letref}\ A\ M)) \qquad A \notin FV(V) & (E_3) \\
(\mathsf{letref}\ A\ M) & = & M \qquad A \notin FV(M) & (E_4) \\
\mathsf{deref} & = & (\lambda x.(\mathsf{deref}\ x)) & (\eta_{v4}) \\
\mathsf{setref!} & = & (\lambda x.\lambda y.((\mathsf{setref!}\ x\ y)\ ;\ ?)) & (\eta_{v5})
\end{array}
$$

Figure 4.7 The Theory λCS for Core Scheme (II).

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 4.6, 4.7, and 4.8. 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. For the simply typed languages, our calculi are also "semantically complete" with respect to denotational CPS models.

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 $D_kE_kF_k$:

$$
\begin{array}{rcll}
(\textsf{if true}\ P\ Q) & = & P & (\mathit{Ifk}_t) \\
(\textsf{if false}\ P\ Q) & = & Q & (\mathit{Ifk}_f) \\
(\textsf{add1}_k\ K\ \ulcorner n \urcorner) & = & (K\ \ulcorner n+1 \urcorner) \quad n \in \mathbb{N} & (\mathit{Addk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner m \urcorner) & = & (K\ \ulcorner n/m \urcorner) \quad m \neq 0 & (\mathit{Divk}) \\
(/_k\ K\ \ulcorner n \urcorner\ \ulcorner 0 \urcorner) & = & \ulcorner 5 \urcorner & (\mathit{Divk}_e) \\
(\textsf{integer?}_k\ K\ \ulcorner n \urcorner) & = & (K\ \textsf{true}) & (\mathit{Intk}_t) \\
(\textsf{integer?}_k\ K\ W) & = & (K\ \textsf{false}) \quad W \neq \ulcorner n \urcorner & (\mathit{Intk}_f) \\
(\textsf{letref}\ A\ (\textsf{setref!}_k\ (\lambda d.P)\ W_1\ W_2)) & = & (\textsf{setref!}_k\ (\lambda d.(\textsf{letref}\ A\ P))\ W_1\ W_2) & (\mathit{Dk}_2) \\
 & & A \notin FV(W_1, W_2), d \notin FV(P) & \\
(\textsf{setref!}_k\ (\lambda d.(\textsf{deref}_k\ K\ A))\ A\ W) & = & (\textsf{setref!}_k\ (\lambda d.KW)\ A\ W) \quad d \notin FV(K) & (\mathit{Ek}_1) \\
(\textsf{setref!}_k\ (\lambda d.(\textsf{deref}_k\ K\ A))\ A_1\ W) & = & (\textsf{deref}_k\ (\lambda x.(\textsf{setref!}_k\ (\lambda d.Kx)\ A_1\ W))A) & (\mathit{Ek}_2) \\
 & & A_1 \neq A, d \notin FV(P) & \\
(\textsf{letref}\ A\ P) & = & P \qquad A \notin FV(P) & (\mathit{Ek}_4)
\end{array}
$$

Figure 4.8 The Theory $\lambda cps(CS)$ for CPS Core Scheme.

4.4 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 [25], to prove the equivalence of two programs, or to simplify a program by a series of meaning-preserving transformations [44]. 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.

4.4.1 The Original Program

For convenience, we use a superset of Core Scheme that includes assignments to variables via **set!**, and various other syntactic extensions [13]:

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

The original definition of coroutines using first-class continuations is [49]:

```
(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 4.9 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.

4.4.2 Simplifying the Program

The transformation of the program proceeds by applying one of the axioms of the **λ***CS*-calculus at each step. For clarity, the redex is surrounded by a box. During

```
(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 4.9 Pseudo Coroutine Code.

the transformation, **set!** is treated as a free variable. Alternatively, we could use the axioms of the λ_v-S-calculus [28] or rewrite the code to use reference cells but this is not necessary for our purposes.

```
(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_{cur}) 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)
                       (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)
                      (f (lambda (dest val)
                           (callcc (lambda (k)
                                     (set! LCS k)
                                     (dest val))))
                         x)
                      (error "fell off end"))])
        (lambda (v) (LCS v)))))
```

4.5 Related Work

The axiomatization of the semantics of call-by-value control operators was originally studied by Felleisen [29, 30] and Talcott [103]. Our axioms extend Felleisen's λ_v-C-calculus and are equivalent to Talcott's theory IOCC (when restricted to our language). Our axiomatization is also the first to be shown complete with respect to the CPS semantics; it was the main technical contribution of our extended journal paper [85]. Independently, Hofmann [52] developed a complete axiomatization of the semantics of call-by-value control operators with respect to CPS semantics; his proof technique is different from ours but the resulting set of axioms is the same (modulo some syntactic differences).

The axiomatization of the semantics of languages with assignments has been studied extensively [8, 17, 23, 26, 29, 51, 61, 102]. Each of the previously proposed logics was designed independently, and differs from the others in subtle ways. Our approach of deriving the calculus from the store-passing axioms is more systematic and generalizes to other languages [88]. Also, when compared to other systems, our calculus has several distinguishing features. First, in contrast with calculi derived from abstract machines for the imperative source language [17, 29, 61], our equational theory never eliminates an expression of the form ($\mathsf{setref!}\ V_1\ V_2$). These expressions simply migrate from point to point within the program to allow lookup operations. In the other calculi [17, 29, 61], there is typically a distinguished position in the program text that contains the most recent value associated with a particular location: setref! expressions modify this current value and lookup operations copy this current value. Our calculus is closer in spirit to the theory $\boldsymbol{\lambda}_{var}$ [73] and the Imperative Lambda Calculus (ILC) [102] but is more powerful because it accommodates first-class reference cells. Also, the theory $\boldsymbol{\lambda}_{var}$ uses a call-by-name semantics for procedures and has an additional **pure** construct. It also uses a "monadic" operator to express the sequencing of assignments while the sequencing of assignments in our case is implicit in the order of evaluation of applications.

Finally, the material in Sections 4.1.4 and 4.1.5 has not appeared elsewhere.

4.6 Summary

We have derived a sound and complete axiomatization of the semantics of call-by-value control operators. Using the store-passing transformation, we have also derived a novel calculus for assignments. Finally, we have developed an equational theory for a realistic language (Core Scheme) that is useful in reasoning about various program properties and that proves the same equations that can be proved after CPS conversion. Our theory eliminates the need for CPS conversion for tools that relied on the stronger equational theory for CPS terms.

Chapter 5

Correspondence of Code Generators

The correspondence in the previous chapters deals with the generation of the intermediate representations in both direct and CPS compilers. In this chapter, we study the operational semantics of these intermediate representations; each operational semantics is expressed using an abstract machine that characterizes the code generator for a typical compiler. Our main result establishes that a realistic code generator for CPS terms is "isomorphic" to the code generator for A-normal form terms. Furthermore, the realistic code generator for CPS terms is derived from the naïve code generator by effectively performing an inverse CPS transformation on the CPS intermediate representation.

The next section reviews the definition of the abstract machine for Core Scheme. Section 2 analyses both naïve and realistic CPS compilers. Section 3 introduces A-normal form compilers. In Section 4, we prove the equivalence between the abstract machines for A-normal form compilers and for realistic CPS compilers. We conclude with a summary of the benefits of using A-normal form terms as an intermediate representation for compilers. Most of the results in this chapter appeared in a paper presented at the Conference on Programming Language Design and Implementation [38].

5.1 Direct Abstract Machine

For the purposes of this chapter, it is sufficient to consider a restriction of Core Scheme that excludes assignments and, for most of the time, control operators. Our language is overly simple but contains all the ingredients that are necessary to generate the result for full core ML or Scheme. In particular, the introduction of assignments, and even control operators, is orthogonal to the analysis of the CPS-based compilation strategy, though we briefly outline the treatment of control operators in Section 5.5. For convenience, we also extend the language with a block-building construct: (**let** $(x\ M)\ N$), which abbreviates the common pattern $((\lambda x.M)\ N)$. Also, for generality, we leave the sets of constants and primitive procedures unspecified, and use multi-argument procedures instead of procedures of one argument:

$$\begin{array}{rcl} M & ::= & V \mid (\mathbf{let}\ (x\ M_1)\ M_2) \mid (\mathbf{if0}\ M_1\ M_2\ M_3) \\ & & \mid (M\ M_1\ \ldots\ M_n) \mid (O\ M_1\ \ldots\ M_n) \end{array}$$

$$\begin{array}{rcl} V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M) \\ c & \in & \mathit{Constants} \\ x & \in & \mathit{Variables} \\ O & \in & \mathit{Primitive\ Operations} \end{array}$$

Since we are interested in modeling the behavior of code generators, we formulate the semantics of the language using an abstract machine that manipulates abstract counterparts to machine stacks, stores, registers, etc. The machine we use, the CEK machine [27], has three components: a control string C that represents the current subexpression of interest, an environment E that includes bindings for all free variables in C, and a continuation K that represents the rest of the computation.

Definition 5.1 *(The CEK-machine for Core Scheme)* The CEK machine specifies the semantics of Core Scheme as follows. Let $M \in CS$,

$$\mathit{eval}_d(M) = c \quad \text{if} \quad \langle M, \emptyset, \mathbf{stop}\rangle \longmapsto^* \langle \mathbf{stop}, c\rangle.$$

State Space:

$$\begin{array}{rcrcl} S & \in & \mathit{State}_d & = & CS \times \mathit{Env}_d \times \mathit{Cont}_d \mid \mathit{Cont}_d \times \mathit{Value}_d \\ E & \in & \mathit{Env}_d & = & \mathit{Variables} \longrightarrow \mathit{Value}_d \\ V^* & \in & \mathit{Value}_d & = & c \mid \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle \\ K & \in & \mathit{Cont}_d & = & \mathbf{stop} \mid \langle \mathbf{ap}\ \langle \ldots, V^*, \bullet, M, \ldots\rangle, E, K\rangle \mid \langle \mathbf{lt}\ x, M, E, K\rangle \\ & & & & \mid \langle \mathbf{if}\ M_1, M_2, E, K\rangle \mid \langle \mathbf{pr}\ O, \langle \ldots, V^*, \bullet, M, \ldots\rangle, E, K\rangle \end{array}$$

Transition Rules:

$$\begin{array}{rcl} \langle V, E, K\rangle & \longmapsto & \langle K, \gamma(V, E)\rangle \\ \langle (\mathbf{let}\ (x\ M_1)\ M_2), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{lt}\ x, M_2, E, K\rangle\rangle \\ \langle (\mathbf{if0}\ M_1\ M_2\ M_3), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{if}\ M_2, M_3, E, K\rangle\rangle \\ \langle (M\ M_1\ \ldots\ M_n), E, K\rangle & \longmapsto & \langle M, E, \langle \mathbf{ap}\ \langle \bullet, M_1, \ldots, M_n\rangle, E, K\rangle\rangle \\ \langle (O\ M_1\ M_2\ \ldots\ M_n), E, K\rangle & \longmapsto & \langle M_1, E, \langle \mathbf{pr}\ O, \langle \bullet, M_2, \ldots, M_n\rangle, E, K\rangle\rangle \end{array}$$

$$
\begin{array}{rcl}
\langle\langle \mathbf{lt}\ x, M, E, K\rangle, V^*\rangle & \longmapsto & \langle M, E[x := V^*], K\rangle \\
\langle\langle \mathbf{if}\ M_1, M_2, E, K\rangle, 0\rangle & \longmapsto & \langle M_1, E, K\rangle \\
\langle\langle \mathbf{if}\ M_1, M_2, E, K\rangle, V^*\rangle & \longmapsto & \langle M_2, E, K\rangle \quad \text{where } V^* \neq 0 \\
\langle\langle \mathbf{ap}\ \langle \ldots, V_i^*, \bullet, M, \ldots\rangle, E, K\rangle, V_{i+1}^*\rangle & \longmapsto & \langle M, E, \langle \mathbf{ap}\ \langle \ldots, V_i^*, V_{i+1}^*, \bullet, \ldots\rangle, E, K\rangle\rangle \\
\langle\langle \mathbf{ap}\ V^*, V_1^*, \ldots, \bullet\rangle, E, K\rangle, V_n^*\rangle & \longmapsto & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], K\rangle \\
 & & \text{if } V^* = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \\
\langle\langle \mathbf{pr}\ O, \langle \ldots, V_i^*, \bullet, M, \ldots\rangle, E, K\rangle, V_{i+1}^*\rangle & \longmapsto & \langle M, E, \langle \mathbf{pr}\ O, \langle \ldots, V_i^*, V_{i+1}^*, \bullet, \ldots\rangle, E, K\rangle\rangle \\
\langle\langle \mathbf{pr}\ O, \langle V_1^*, \ldots, \bullet\rangle, E, K\rangle, V_n^*\rangle & \longmapsto & \langle K, \delta(O, V_1^*, \ldots, V_n^*)\rangle \\
 & & \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined}
\end{array}
$$

Auxiliary Functions:

$$
\begin{array}{rcl}
\gamma(c, E) & = & c \\
\gamma(x, E) & = & E(x) \\
\gamma((\lambda x_1 \ldots x_n.M), E) & = & \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle
\end{array}
$$

The CEK machine changes state according to the transition function in the above definition. For example, the state transition for the expression $((\lambda x.x)\ (\lambda y.y))$ starts the evaluation of $(\lambda x.x)$ in the current environment E and modifies the continuation register to encode the rest of the computation $\langle \mathbf{fun}\ (\lambda y.y), E, K\rangle$. When the new continuation receives a value, it evaluates the argument $(\lambda y.y)$ and modifies the continuation register appropriately. The remaining clauses have similarly intuitive explanations.

The relation $\longmapsto^*$ is the reflexive transitive closure of the transition function. The function γ constructs machine values from syntactic values and environments. The notation $E(x)$ refers to an algorithm for looking up the value of x in the environment E. The operation $E[x_1 := V_1^*, \ldots, x_n := V_n^*]$ extends the environment E such that subsequent lookups of x_i return the value V_i^*. The object $\langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle$ is a *closure*, a record that contains the code for M and values for the free variables of $(\lambda x_1 \ldots x_n.M)$. The partial function δ abstracts the semantics of the primitive operations.

The CEK machine provides a model for designing direct compilers [11, 31, 59]. A compiler based on the CEK machine implements an efficient representation for environments, *e.g.*, displays, and for continuations, *e.g.*, a stack. The machine code produced by such a compiler realizes the abstract operations specified by the CEK machine by manipulating these concrete representations of environments and continuations.

5.2 CPS Abstract Machine

A CPS program uses continuation expressions to encode the rest of the computation, thus shifting the burden of maintaining control information from the abstract machine to the code. The CPS intermediate language used in realistic CPS compilers [3, 56, 97] corresponds to the output of our compacting CPS transformation. For the specific language of interest, we get the language $cps(CS)$.[13]

Definition 5.2 *(The language cps(CS))* The set of terms is inductively defined as follows:

$$
\begin{array}{rcll}
P & ::= & (k\ W) & (\mathit{return}) \\
 & & \mid (\mathbf{let}\ (x\ W)\ P) & (\mathit{bind}) \\
 & & \mid (\mathbf{if0}\ W\ P_1\ P_2) & (\mathit{branch}) \\
 & & \mid (W\ k\ W_1\ \ldots\ W_n) & (\mathit{tail\ call}) \\
 & & \mid (W\ (\lambda x.P)\ W_1\ \ldots\ W_n) & (\mathit{call}) \\
 & & \mid (O_k\ k\ W_1\ \ldots\ W_n) & (\mathit{prim\text{-}op}) \\
 & & \mid (O_k\ (\lambda x.P)\ W_1\ \ldots\ W_n) & (\mathit{prim\text{-}op}) \\
W & ::= & c \mid x \mid (\lambda k x_1 \ldots x_n.P) & (\mathit{values})
\end{array}
$$

Naïve CPS Compilers The abstract machine that characterizes the code generator of a naïve CPS compiler is the $\mathrm{C_{cps}E}$ machine. The machine is the obvious restriction of the CEK-machine to the set of CPS terms. In particular, the machine does not require a continuation component (K) to record the rest of the computation since terms in $cps(CS)$ contain an encoding of control-flow information.

Definition 5.3 *(The naïve CPS abstract machine: the $C_{cps}E$ machine)* The machine specifies the semantics of $cps(CS)$ as follows:

$$eval_n(P) = c \quad \text{if} \quad \langle P, \emptyset[k := \langle \mathbf{cl}\ x, (k\ x), \emptyset[k := \mathbf{stop}]\rangle]\rangle \longmapsto^* \langle (k\ x), \emptyset[x := c, k := \mathbf{stop}]\rangle.$$

State Space:

$$
\begin{array}{rcrcl}
S_n & \in & State_n & = & cps(CS) \times Env_n \\
E & \in & Env_n & = & Variables \multimap Value_n \\
W^* & \in & Value_n & = & c \mid \langle \mathbf{cl}\ kx_1 \ldots x_n, P, E\rangle \mid \langle \mathbf{cl}\ x, P, E\rangle \mid \mathbf{stop}
\end{array}
$$

[13]Since our language accommodates multi-argument procedures, we modify the CPS transformation slightly to pass all arguments simultaneously.

Transition Rules:

$$
\begin{array}{rcl}
\langle (k\ W), E\rangle & \longmapsto & \langle P', E'[x := \mu(W,E)]\rangle \\
 & & \text{where } E(k) = \langle \mathbf{cl}\ x, P', E'\rangle \\
\langle (\mathbf{let}\ (x\ W)\ P), E\rangle & \longmapsto & \langle P, E[x := \mu(W,E)]\rangle \\
\langle (\mathbf{if0}\ W\ P_1\ P_2), E\rangle & \longmapsto & \langle P_1, E\rangle \quad \text{where } \mu(W,E) = 0 \\
 & \text{or} & \langle P_2, E\rangle \quad \text{where } \mu(W,E) \neq 0 \\
\langle (W\ k\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[k' := E(k), x_1 := W_1^*, \ldots, x_n := W_n^*]\rangle \\
 & & \text{where } \mu(W,E) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E'\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (W\ (\lambda x.P)\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[k' := \langle \mathbf{cl}\ x, P, E\rangle, x_1 := W_1^*, \ldots]\rangle \\
 & & \text{where } \mu(W,E) = \langle \mathbf{cl}\ k'x_1 \ldots x_n, P', E'\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (O_k\ k\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P', E'[x := \delta_c(O_k, W_1^*, \ldots, W_n^*)]\rangle \\
 & & \text{if } \delta_c(O_k, W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{where } E(k) = \langle \mathbf{cl}\ x, P', E'\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E) \\
\langle (O_k\ (\lambda x.P)\ W_1\ \ldots\ W_n), E\rangle & \longmapsto & \langle P, E[x := \delta_c(O_k, W_1^*, \ldots, W_n^*)]\rangle \\
 & & \text{if } \delta_c(O_k, W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E)
\end{array}
$$

Auxiliary Functions:

$$
\begin{array}{rcl}
\mu(c, E) & = & c \\
\mu(x, E) & = & E(x) \\
\mu((\lambda k x_1 \ldots x_n.P), E) & = & \langle \mathbf{cl}\ k x_1 \ldots x_n, P, E\rangle
\end{array}
$$

Realistic CPS Compilers Although the $\mathrm{C_{cps}E}$ machine describes how the object code of a naïve compiler would be executed, realistic compilers deviate from this model in two regards.

First, the naïve abstract machine for CPS code represents the continuation as an ordinary closure. Yet, realistic CPS compilers "mark" the continuation closure as a special closure. For example, Shivers partitions procedures and continuations in order to improve the data flow analysis of CPS programs [92, sec 3.8.3]. Also, in both Orbit [56] and Rabbit [97], the allocation strategy of a closure changes if the closure is a continuation. Similarly, Appel [3, p 114–124] describes various techniques for closure allocation that treat the continuation closure in a special way.

In order to reflect these changes in the machine, we tag continuation closures with a special marker "**ar**" that describes them as **a**ctivation **r**ecords.

Second, the CPS representation of any user-defined procedure receives a continuation argument. However, Steele [97] modifies the CPS transformation with a "continuation variable hack" [97, p 94] that recognizes instances of CPS terms like $((\lambda k_1 \cdots .P)\ k_2\ \ldots)$ and transforms them to $((\lambda \cdots .P[k_1 := k_2])\ \ldots)$. This "optimization" eliminates "some of the register shuffling" [97, p 94] during the evaluation of the term. Appel [3] achieves the same effect without modifying the CPS transformation by letting the variables k_1 and k_2 *share* the same register during the procedure call.

In terms of the CPS abstract machine, the optimization corresponds to a modification of the operation $E'[k' := E(k), x_1 := W_1^*, \ldots, x_n := W_n^*]$ to $E'[x_1 := W_1^*, \ldots, x_n := W_n^*]$ such that E and E' share the binding of k. In order to make the sharing explicit, we split the environment into two components: a component E^k that includes the binding for the continuation, and a component E^- that includes the rest of the bindings, and treat each component independently. This optimization relies on the fact that every control string has exactly one free continuation variable, which implies that the corresponding value can be held in a special register.[14]

Performing these modifications on the naïve abstract machine produces the realistic CPS abstract machine.

Definition 5.4 *(The realistic CPS abstract machine: the $C_{cps}EK$ machine)* The machine specifies the semantics of $cps(CS)$ as follows. Let $P \in cps(CS)$,

$$eval_c(P) = c \quad \text{if} \quad \langle P, \emptyset, \langle \mathbf{ar}\ x, (k\ x), \emptyset, \mathbf{stop}\rangle\rangle \longmapsto_c^* \langle (k\ x), \emptyset[x := c], \mathbf{stop}\rangle.$$

State Space:

$$\begin{array}{rcrcl}
S_c & \in & State_c & = & cps(CS) \times Env_c \times Cont_c \\
E^- & \in & Env_c & = & Variables \longrightarrow\!\!\!\!\!\!\circ\;\; Value_c \\
W^* & \in & Value_c & = & c \mid \langle \mathbf{cl}\ kx_1 \ldots x_n, P, E^- \rangle \\
E^k & \in & Cont_c & = & \mathbf{stop} \mid \langle \mathbf{ar}\ x, P, E^-, E^k \rangle
\end{array}$$

[14]This fact also holds in the presence of control operators as there is always one identifiable *current* continuation.

Transition Rules:

$$
\begin{array}{rcl}
\langle (k\ W), E^-, E^k \rangle & \xrightarrow{(1)}_c & \langle P', E_1^-[x := \mu(W, E^-)], E_1^k \rangle \\
 & & \text{where } E^k = \langle \textbf{ar}\ x, P', E_1^-, E_1^k \rangle \\
\langle (\textbf{let}\ (x\ W)\ P), E^-, E^k \rangle & \xrightarrow{(2)}_c & \langle P, E^-[x := \mu(W, E^-)], E^k \rangle \\
\langle (\textbf{if0}\ W\ P_1\ P_2), E^-, E^k \rangle & \xrightarrow{(3)}_c & \langle P_1, E^-, E^k \rangle \quad \text{where } \mu(W, E^-) = 0 \\
 & \text{or} & \langle P_2, E^-, E^k \rangle \quad \text{where } \mu(W, E^-) \neq 0 \\
\langle (W\ k\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \xrightarrow{(4)}_c & \langle P', E_1^-[x_1 := W_1^*, \ldots, x_n := W_n^*], E^k \rangle \\
 & & \text{where } \mu(W, E^-) = \langle \textbf{cl}\ k' x_1 \ldots x_n, P', E_1^- \rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (W\ (\lambda x.P)\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \xrightarrow{(5)}_c & \langle P', E_1^-[x_1 := W_1^*, \ldots, x_n := W_n^*], E'^k \rangle \\
 & & \text{where } \mu(W, E^-) = \langle \textbf{cl}\ k' x_1 \ldots x_n, P', E_1^- \rangle \\
 & & \text{and } E'^k = \langle \textbf{ar}\ x, P, E^-, E^k \rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (O_k\ k\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \xrightarrow{(6)}_c & \langle P', E_1^-[x := \delta_c(O_k, W_1^*, \ldots, W_n^*)], E_1^k \rangle \\
 & & \text{if } \delta_c(O_k, W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{where } E^k = \langle \textbf{ar}\ x, P', E_1^-, E_1^k \rangle \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-) \\
\langle (O_k\ (\lambda x.P)\ W_1\ \ldots\ W_n), E^-, E^k \rangle & \xrightarrow{(7)}_c & \langle P, E^-[x := \delta_c(O_k, W_1^*, \ldots, W_n^*)], E^k \rangle \\
 & & \text{if } \delta_c(O_k, W_1^*, \ldots, W_n^*) \text{ is defined,} \\
 & & \text{and for } 1 \leq i \leq n,\ W_i^* = \mu(W_i, E^-)
\end{array}
$$

The new $\mathrm{C_{cps}EK}$ machine extracts the information regarding the continuation from the CPS terms and manages the continuation in an optimized way. For example, the state transition for the tail call $(W\ k\ W_1\ \ldots\ W_n)$ evaluates W to a closure $\langle \textbf{cl}\ kx_1 \ldots x_n, P', E_1^- \rangle$, extends E_1^- with the values of $W_1, \ldots, W_n$ and starts the execution of P'. In particular, there is no need to extend E_1^- with the value of k as this value remains in the environment component E^k.

5.3 A-Normal Forms as an Intermediate Language

A close inspection of the $\mathrm{C_{cps}EK}$ machine reveals that the control strings often contain redundant information considering the way instructions are executed. First, a *return* instruction, *i.e.*, the transition $\xrightarrow{(1)}_c$, dispatches on the term $(k\ W)$, which informs the machine that the "return address" is denoted by the value of the variable k. The machine ignores this information since a *return* instruction automatically uses the value of register E^k as the "return address". Second, the *call* instructions, *i.e.*, transitions $\xrightarrow{(4)}_c$ and $\xrightarrow{(5)}_c$, invoke closures that expect, among other arguments, a continuation k. Again, the machine ignores the continuation parameter in the closures and manipulate the "global" register E^k instead.

The crucial insight is that the elimination of the redundant information from the $C_{cps}EK$ machine corresponds to an inverse CPS transformation on the intermediate code. The function $\mathcal{C}^{-1}$ realizes such an inverse; its output $A(CS)$ is the "effective" intermediate language of realistic CPS compilers.

Definition 5.5 *(The language $A(CS)$)* The language $A(CS)$ is produced by the natural inverse CPS transformation:

$$\begin{array}{rcl}
\mathcal{C}^{-1} : cps(CS) & \rightarrow & A(CS) \\
\mathcal{C}^{-1}[\![(k\ W)]\!] & = & \Phi^{-1}[\![W]\!] \\
\mathcal{C}^{-1}[\![(\mathbf{let}\ (x\ W)\ P)]\!] & = & (\mathbf{let}\ (x\ \Phi^{-1}[\![W]\!])\ \mathcal{C}^{-1}[\![P]\!]) \\
\mathcal{C}^{-1}[\![(\mathbf{if0}\ W\ P_1\ P_2)]\!] & = & (\mathbf{if0}\ \Phi^{-1}[\![W]\!]\ \mathcal{C}^{-1}[\![P_1]\!]\ \mathcal{C}^{-1}[\![P_2]\!]) \\
\mathcal{C}^{-1}[\![(W\ k\ W_1\ \ldots\ W_n)]\!] & = & (\Phi^{-1}[\![W]\!]\ \Phi^{-1}[\![W_1]\!]\ \ldots\ \Phi^{-1}[\![W_n]\!]) \\
\mathcal{C}^{-1}[\![(W\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & (\mathbf{let}\ (x\ (\Phi^{-1}[\![W]\!]\ \Phi^{-1}[\![W_1]\!]\ \ldots\ \Phi^{-1}[\![W_n]\!]))\ \mathcal{C}^{-1}[\![P]\!]) \\
\mathcal{C}^{-1}[\![(O_k\ k\ W_1\ \ldots\ W_n)]\!] & = & (O\ \Phi^{-1}[\![W_1]\!]\ \ldots\ \Phi^{-1}[\![W_n]\!]) \\
\mathcal{C}^{-1}[\![(O_k\ (\lambda x.P)\ W_1\ \ldots\ W_n)]\!] & = & (\mathbf{let}\ (x\ (O\ \Phi^{-1}[\![W_1]\!]\ \ldots\ \Phi^{-1}[\![W_n]\!]))\ \mathcal{C}^{-1}[\![P]\!])
\end{array}$$

$$\begin{array}{rcl}
\Phi^{-1} : W & \rightarrow & V \\
\Phi^{-1}[\![c]\!] & = & c \\
\Phi^{-1}[\![x]\!] & = & x \\
\Phi^{-1}[\![\lambda k x_1 \ldots x_n.P]\!] & = & \lambda x_1 \ldots x_n.\mathcal{C}^{-1}[\![M]\!]
\end{array}$$

The set of terms is inductively defined as follows:

$$\begin{array}{rcll}
M & ::= & V & (\mathit{return}) \\
& & \mid (\mathbf{let}\ (x\ V)\ M) & (\mathit{bind}) \\
& & \mid (\mathbf{if0}\ V\ M\ M) & (\mathit{branch}) \\
& & \mid (V\ V_1\ \ldots\ V_n) & (\mathit{tail\ call}) \\
& & \mid (\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M) & (\mathit{call}) \\
& & \mid (O\ V_1\ \ldots\ V_n) & (\mathit{prim\text{-}op}) \\
& & \mid (\mathbf{let}\ (x\ (O\ V_1\ \ldots\ V_n))\ M) & (\mathit{prim\text{-}op})
\end{array}$$

$$\begin{array}{rcll}
V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M) & (\mathit{values})
\end{array}$$

Based on this argument, it appears that CPS compilers perform a sequence of three steps:

$$\begin{tikzcd}[column sep=large, row sep=large]
CS \arrow[r, "\text{CPS}"] \arrow[d, dashed, "A\text{-normalization}"'] & \bullet \arrow[d, "\overline{\beta}\text{-normalization}"] \\
A(CS) & cps(CS) \arrow[l, "\text{un-CPS}"']
\end{tikzcd}$$

As argued in Section 3.1, this diagram naturally suggests a direct translation that corresponds to the elimination of the CPS administrative reductions; in our case this translation can be accomplished using the following A-reductions:

$$\begin{array}{rcll}
E[(\mathbf{let}\ (x\ M)\ N)] & \longrightarrow & (\mathbf{let}\ (x\ M)\ E[N])\quad \text{where } E \neq [\,], x \notin FV(E) & (\beta_{\mathit{lift}}) \\
E[(\mathbf{if0}\ V\ M_1\ M_2)] & \longrightarrow & (\mathbf{if0}\ V\ E[M_1]\ E[M_2])\quad \text{where } E \neq [\,] & (\mathit{If}_{\mathit{lift}}) \\
 & & & \\
E[(F\ V_1\ \ldots\ V_n)] & \longrightarrow & (\mathbf{let}\ (x\ (F\ V_1\ \ldots\ V_n))\ E[x]) & (\beta'_{\Omega}) \\
 & & \text{where } F = V \text{ or } F = O, & \\
 & & E \neq E'[(\mathbf{let}\ (z\ [\,])\ M)], E \neq [\,], x \notin FV(E) &
\end{array}$$

In practice, the reduction $\mathit{If}_{\mathit{lift}}$ is unacceptable because it duplicates the evaluation context around the conditional, which may result in an exponential code explosion. With a slight change in the CPS transformation (cf. Page 89), we can eliminate this reduction without affecting any of our results. In that case, the A-normalization phase can be performed using the linear algorithm in Figure 5.1. The algorithm is written in Scheme extended with a special form **match**, which performs pattern matching on the syntax of program terms [105]. It employs a programming technique for CPS algorithms pioneered by Danvy and Filinski [21]. We assume the front-end uniquely renames all variables, which implies that the condition $x \notin FV(E)$ of the reduction β_{lift} holds.

5.4 Equivalence of Abstract Machines

In order to establish that the A-reductions generate the actual intermediate code of CPS compilers, we design an abstract machine for the language of A-normal forms, the C_aEK machine, and prove that this machine is "equivalent" to the CPS machine in Definition 5.4.

```
(define normalize-term (lambda (M) (normalize M (lambda (x) x))))

(define normalize
  (lambda (M k)
    (match M
      [`(lambda ,params ,body) (k `(lambda ,params ,(normalize-term body)))]
      [`(let (,x ,M1) ,M2) (normalize M1
                              (lambda (N1) `(let (,x ,N1) ,(normalize M2 k))))]
      [`(if0 ,M1 ,M2 ,M3) (normalize-name M1
                              (lambda (t)
                                (k `(if0 ,t ,(normalize-term M2) ,(normalize-term M3)))))]
      [`(,Fn . ,M*) (if (PrimOp? Fn)
                        (normalize-name* M* (lambda (t*) (k `(,Fn . ,t*))))
                        (normalize-name Fn
                          (lambda (t)
                            (normalize-name* M* (lambda (t*) (k `(,t . ,t*)))))))]
      [V (k V)])))

(define normalize-name
  (lambda (M k)
    (normalize M (lambda (N)
                   (if (Value? N)
                       (k N)
                       (let ([t (newvar)]) `(let (,t ,N) ,(k t))))))))

(define normalize-name*
  (lambda (M* k)
    (if (null? M*)
        (k '())
        (normalize-name (car M*)
          (lambda (t) (normalize-name* (cdr M*) (lambda (t*) (k `(,t . ,t*)))))))))
```

Figure 5.1 A linear-time A-normalization algorithm

Definition 5.6 *(The C_aEK machine)* The machine specifies the semantics of $A(CS)$ as follows. Let $M \in A(CS)$,

$$eval_a(M) = c \quad \text{if} \quad \langle M, \emptyset, \langle \mathbf{ar}\ x, x, \emptyset, \mathbf{stop}\rangle\rangle \longmapsto_a^* \langle x, \emptyset[x := c], \mathbf{stop}\rangle.$$

State Space:

$$
\begin{array}{rcrcl}
S_a & \in & State_a & = & A(CS) \times Env_a \times Cont_a \\
E & \in & Env_a & = & Variables \longrightarrow\!\!\!\!\!\circ\; Value_a \\
V^* & \in & Value_a & = & c \mid \langle \mathbf{cl}\ x_1 \ldots x_n, M, E\rangle \qquad \text{(machine values)} \\
K & \in & Cont_a & = & \mathbf{stop} \mid \langle \mathbf{ar}\ x, M, E, K\rangle
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcl}
\langle V, E, K\rangle & \xrightarrow{(1)}_a & \langle M', E'[x := \gamma(V,E)], K'\rangle \\
 & & \text{where } K = \langle \mathbf{ar}\ x, M', E', K'\rangle \\
\langle (\mathbf{let}\ (x\ V)\ M), E, K\rangle & \xrightarrow{(2)}_a & \langle M, E[x := \gamma(V,E)], K\rangle \\
\langle (\mathbf{if0}\ V\ M_1\ M_2), E, K\rangle & \xrightarrow{(3)}_a & \langle M_1, E, K\rangle \quad \text{where } \gamma(V,E) = 0 \\
 & \text{or} & \langle M_2, E, K\rangle \quad \text{where } \gamma(V,E) \neq 0 \\
\langle (V\ V_1\ \ldots\ V_n), E, K\rangle & \xrightarrow{(4)}_a & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], K\rangle \\
 & & \text{where } \gamma(V,E) = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle (\mathbf{let}\ (x\ (V\ V_1\ \ldots\ V_n))\ M), E, K\rangle & \xrightarrow{(5)}_a & \langle M', E'[x_1 := V_1^*, \ldots, x_n := V_n^*], K'\rangle \\
 & & \text{where } \gamma(V,E) = \langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle \\
 & & \text{and } K' = \langle \mathbf{ar}\ x, M, E, K\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle (O\ V_1\ \ldots\ V_n), E, K\rangle & \xrightarrow{(6)}_a & \langle M', E'[x := \delta(O, V_1^*, \ldots, V_n^*)], K'\rangle \\
 & & \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined,} \\
 & & \text{where } K = \langle \mathbf{ar}\ x, M', E', K'\rangle \\
 & & \text{and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E) \\
\langle (\mathbf{let}\ (x\ (O\ V_1\ \ldots\ V_n))\ M), E, K\rangle & \xrightarrow{(7)}_a & \langle M, E[x := \delta(O, V_1^*, \ldots, V_n^*)], K\rangle \\
 & & \text{if } \delta(O, V_1^*, \ldots, V_n^*) \text{ is defined,} \\
 & & \text{and for } 1 \leq i \leq n,\ V_i^* = \gamma(V_i, E)
\end{array}
$$

The C_aEK machine is a CEK machine specialized to the subset of Core Scheme in A-normal form (Definition 5.5). The machine (see Definition 5.6) has only two kinds of continuations: the continuation **stop**, and continuations of the form $\langle \mathbf{ar}\ x, M, E, K\rangle$. Unlike the CEK machine, the C_aEK machine only needs to build a continuation for the evaluation of a non-tail function call. For example, the transition rule for the tail call $(V\ V_1\ \ldots\ V_n)$ evaluates V to a closure $\langle \mathbf{cl}\ x_1 \ldots x_n, M', E'\rangle$, extends the environment E' with the values of $V_1, \ldots, V_n$ and continues with the execution of M'. The continuation component remains in the register K. By comparison, the CEK machine would build a separate continuation for the evaluation of each subexpression $V, V_1, \ldots, V_n$.

A comparison of Definitions 5.4 and 5.6 suggests a close relationship between the $\mathrm{C}_{\mathrm{cps}}$EK machine and the C_aEK machine. In fact, the two machines are identical modulo the *syntax* of the control strings, as corresponding state transitions on the two machines perform the same abstract operations. Currently, the transition rules for these machines are defined using pattern matching on the syntax of terms. Once we reformulate these rules using *predicates* and *selectors* for abstract syntax, we can see the correspondence more clearly.

For example, we can abstract the transition rules $\xmapsto{(2)}_a$ and $\xmapsto{(2)}_c$ from the term syntax as the higher-order function $\mathcal{T}_2$:

$\mathcal{T}_2[\mathit{bind?}, \mathit{bind\text{-}var}, \mathit{bind\text{-}body}, \mathit{bind\text{-}val}] =$

$$\langle C, E, K\rangle \longmapsto \langle \mathit{bind\text{-}body}(C), E[\mathit{bind\text{-}var}(C) := \alpha(\mathit{bind\text{-}val}(C))], K\rangle$$

if $\mathit{bind?}(C)$ and where α is the appropriate conversion from syntactic values to machine values. The arguments to $\mathcal{T}_2$ are abstract-syntax functions for manipulating terms in a syntax-independent manner. Applying $\mathcal{T}_2$ to the appropriate functions produces either the transition rule $\xmapsto{(2)}_a$ of the $\mathrm{C}_a\mathrm{EK}$ machine or the rule $\xmapsto{(2)}_c$ of the $\mathrm{C}_{\mathrm{cps}}\mathrm{EK}$ machine, *i.e.*,

$$\begin{aligned}
\xmapsto{(2)}_a &= \mathcal{T}_2[\mathit{A\text{-}bind?}, \mathit{A\text{-}bind\text{-}var}, \mathit{A\text{-}bind\text{-}body}, \mathit{A\text{-}bind\text{-}val}]\\
\xmapsto{(2)}_c &= \mathcal{T}_2[\mathit{cps\text{-}bind?}, \mathit{cps\text{-}bind\text{-}var}, \mathit{cps\text{-}bind\text{-}body}, \mathit{cps\text{-}bind\text{-}val}]
\end{aligned}$$

Suitable definitions of the syntax-functions for the language $A(CS)$ are:

$$\begin{aligned}
\mathit{A\text{-}bind?}[\![(\mathbf{let}\ (x\ V)\ M)]\!] &= \mathit{true}\\
\mathit{A\text{-}bind\text{-}var}[\![(\mathbf{let}\ (x\ V)\ M)]\!] &= x\\
\mathit{A\text{-}bind\text{-}body}[\![(\mathbf{let}\ (x\ V)\ M)]\!] &= M\\
\mathit{A\text{-}bind\text{-}val}[\![(\mathbf{let}\ (x\ V)\ M)]\!] &= V
\end{aligned}$$

Definitions for the language $\mathit{cps}(CS)$ follow a similar pattern:

$$\begin{aligned}
\mathit{cps\text{-}bind?}[\![(\mathbf{let}\ (x\ W)\ P)]\!] &= \mathit{true}\\
\mathit{cps\text{-}bind\text{-}var}[\![(\mathbf{let}\ (x\ W)\ P)]\!] &= x\\
\mathit{cps\text{-}bind\text{-}body}[\![(\mathbf{let}\ (x\ W)\ P)]\!] &= P\\
\mathit{cps\text{-}bind\text{-}val}[\![(\mathbf{let}\ (x\ W)\ P)]\!] &= W
\end{aligned}$$

In the same manner, we can abstract each pair of transition rules $\xmapsto{(n)}_a$ and $\xmapsto{(n)}_c$ as a higher-order functional $\mathcal{T}_n$. This argument shows that the transitions functions of the $\mathrm{C}_a\mathrm{EK}$ and $\mathrm{C}_{\mathrm{cps}}\mathrm{EK}$ machines are identical modulo syntax. However, in order to show that the evaluation of an A-normal form term M and its CPS counterpart on the respective machines produces exactly the same behavior, we also need to prove that there exists a bijection $\mathcal{M}$ between machine states that commutes with the transition rules.

Definition 5.7 *($\mathcal{M}$, $\mathcal{R}$, $\mathcal{V}$, and $\mathcal{K}$)* The following functions map the states of the $\mathrm{C_{cps}EK}$ machine to the states of the $\mathrm{C}_a\mathrm{EK}$ machine:

$$\mathcal{M} : State_c \longrightarrow State_a$$
$$\mathcal{M}(\langle P, E^-, E^k \rangle) = \langle \mathcal{C}^{-1}[\![P]\!], \mathcal{R}(E^-), \mathcal{K}(E^k) \rangle$$

$$\mathcal{R} : Env_c \longrightarrow Env_a$$
$$\mathcal{R}(E^-) = E \qquad \text{where } E(x) = \mathcal{V}(E^-(x))$$

$$\mathcal{V} : Value_c \longrightarrow Value_a$$
$$\mathcal{V}(c) = c$$
$$\mathcal{V}(\langle \mathbf{cl}\ kx_1 \ldots x_n, P, E^- \rangle) = \langle \mathbf{cl}\ x_1 \ldots x_n, \mathcal{C}^{-1}[\![P]\!], \mathcal{R}(E^-) \rangle$$

$$\mathcal{K} : Cont_c \longrightarrow Cont_a$$
$$\mathcal{K}(\mathbf{stop}) = \mathbf{stop}$$
$$\mathcal{K}(\langle \mathbf{ar}\ x, P, E^-, E^k \rangle) = \langle \mathbf{ar}\ x, \mathcal{C}^{-1}[\![P]\!], \mathcal{R}(E^-), \mathcal{K}(E^k) \rangle$$

Intuitively, the function $\mathcal{M}$ maps $\mathrm{C_{cps}EK}$ machine states to $\mathrm{C}_a\mathrm{EK}$ machine states, and $\mathcal{R}$, $\mathcal{V}$ and $\mathcal{K}$ perform a similar mapping for environments, machine values and continuations respectively. We can now formalize the correspondence between the machines assuming that δ and δ_c behave as expected.

Definition 5.8 *(Good δ; Good δ_c)* The functions δ and δ_c are *good* if and only if the following condition holds for all $W_1^*, \ldots, W_n^* \in Value_c$:

$$\mathcal{V}(\delta_c(O_k, W_1^*, \ldots, W_n^*)) = \delta(O, \mathcal{V}(W_1^*), \ldots, \mathcal{V}(W_n^*))$$

Theorem 5.1 *(Commutativity Theorem)* Let $S \in State_c$. If δ and δ_c are *good*, then $S \overset{(n)}{\longmapsto}_c S'$ if and only if $\mathcal{M}(S) \overset{(n)}{\longmapsto}_a \mathcal{M}(S')$.

$$\begin{tikzcd}
S \arrow[r, phantom, "\overset{(n)}{\longmapsto}_c"] \arrow[d, dashed, <->, "\mathcal{M}"'] & S' \arrow[d, dashed, <->, "\mathcal{M}"'] \\
\mathcal{M}(S) \arrow[r, phantom, "\overset{(n)}{\longmapsto}_a"] & \mathcal{M}(S')
\end{tikzcd}$$

Proof The inverse CPS transformations $\mathcal{C}^{-1}$ is bijective. Hence by structural induction, the functions $\mathcal{M}$, $\mathcal{R}$, $\mathcal{V}$ and $\mathcal{K}$ are also bijective. Using this fact, the proof proceeds by case analysis on each of the machine transition functions. $\square$

Intuitively, the evaluation of a CPS term P on the $\mathrm{C_{cps}EK}$ machine proceeds in the same fashion as the evaluation of $\mathcal{C}^{-1}[\![P]\!]$ on the $\mathrm{C}_a\mathrm{EK}$ machine. This implies that both machines perform the same sequence of abstract operations, and hence compilers based on these abstract machines can produce identical code for the same input. The A-normal form compiler achieves its goal in fewer passes.

5.5 Control Operators

We illustrate how to extend our result to languages with control operators like *callcc*. Since the CPS transform of *callcc*:

$$\Phi[\![callcc]\!] = \lambda ku.(u\ k\ (\lambda dx.kx))$$

does not extend the CPS language with new kinds of terms, the idealized CPS abstract machine is *not* affected by the addition of *callcc*. However, realistic CPS compilers deviate from the theoretical CPS abstract machine and interpret this intermediate code on a more specialized machine:

- Orbit [56] essentially ignores the operator *callcc* "because the runtime system ensures that any continuation which is captured by `call-with-current-continuation` will be migrated dynamically to the heap" [56, p. 227].

- SML-NJ [3] ensures that the current continuation is always accessible in a register. In the absence of control operators, the variable k in the expression $(k\ W)$ always refers to the current continuation and is therefore useless. However, once the language allows continuations to be captured, the variable k may refer to a previously captured continuation. The compiler does *not* guarantee that this latter continuation is in a register.

Both compilers differentiate between "regular" continuations and "captured" continuations. Thus, we modify the syntax of CPS terms accordingly:

$$
\begin{array}{rcll}
P & ::= & (k\ W) & (\mathit{return} : k \text{ is the current continuation}) \\
 & & \mid (@k\ W) & (\mathit{jump} : k \text{ is a captured continuation}) \\
 & & \mid \cdots & \\
W & ::= & \cdots \mid \lambda kx.P \mid \lambda @kx.P &
\end{array}
$$

The "realistic" CPS abstract machine treats captured continuations in a special way. The additional transitions are in Figure 5.2.

$$
\begin{array}{rcl}
\langle (@k\ W), E^-, E^k \rangle & \longmapsto & \langle P', E_1^-[x := \gamma(W, E^-)], E_1^k \rangle \\
 & & \text{where } E^-(k) = \langle \mathbf{ar}\ , x, P', E_1^-, E_1^k \rangle \\
\langle (W\ k\ W_1\ \cdots\ W_n), E^-, E^k \rangle & \longmapsto & \langle P', E_1^-[x_1 := W_1^*, \cdots, k' := E^k], E^k \rangle \\
 & & \text{where } \gamma(W, E^-) = \langle \mathbf{cl}\ @k'x_1 \ldots x_n, P', E_1^- \rangle \\
 & & \text{and for } 1 \le n, W_i^* = \gamma(W_i, E^-) \\
\langle (W\ (\lambda x.P)\ W_1\ \cdots\ W_n), E^-, E^k \rangle & \longmapsto & \langle P', E_1^-[x_1 := W_1^*, \cdots, k' := K'], K' \rangle \\
 & & \text{where } \gamma(W, E^-) = \langle \mathbf{cl}\ @k'x_1 \ldots x_n, P', E_1^- \rangle \\
 & & K' = \langle \mathbf{ar}\ x, P, E^-, E^k \rangle \\
 & & \text{and for } 1 \le i \le n, W_i^* = \gamma(W_i, E^-)
\end{array}
$$

Figure 5.2 Extending the realistic CPS machine with *callcc*

It is straightforward to modify the language of A-normal forms accordingly:

$$
\begin{array}{rcl}
M & ::= & \cdots \mid (@c\ V) \\
V & ::= & \cdots \mid (\lambda x_1 \ldots x_n.(\mathit{callcc}\ (\lambda @c.M)))
\end{array}
$$

The corresponding transitions for the abstract machine are in Figure 5.3. The transitions correspond to the transitions of the "real" CPS abstract machine.

5.6 Summary

The technique of A-normalization provides an organizational principle for the construction of compilers that combines various stages of fully developed CPS compilers in one straightforward transformation. Our analysis suggests that the language of A-normal forms is a good intermediate representation for compilers. Indeed, most

$$
\begin{array}{rcl}
\langle (@c\ V), E, K \rangle & \longrightarrow & \langle M', E'[x := \gamma(V, E)], K' \rangle \\
& & \text{where } E(c) = \langle \mathbf{ar}\ x, E', M', K' \rangle \\
\langle (V\ V_1\ \cdots\ V_n), E, K \rangle & \longrightarrow & \langle M', E'[x_1 := V_1^*, \cdots, x_n := V_n^*, c := K], K \rangle \\
& & \text{where } \gamma(V, E) = \langle \mathbf{cl}\ x_1 \ldots x_n, B, E' \rangle \\
& & \text{and } B = (\mathit{callcc}\ (\lambda @c.M')) \\
& & \text{and for } 1 \leq i \leq n, V_i^* = \gamma(V_i, E) \\
\langle (\mathbf{let}\ (x\ (V\ V_1\ \cdots\ V_n))\ M), E, K \rangle & \longrightarrow & \langle M', E'[x_1 := V_1^*, \cdots, x_n := V_n^*, c := K'], K' \rangle \\
& & \text{where } \gamma(V, E) = \langle \mathbf{cl}\ x_1 \ldots x_n, B, E' \rangle \\
& & \text{and } B = (\mathit{callcc}\ (\lambda @c.M')) \\
& & \text{and } K' = \langle \mathbf{ar}, x, E, M, K \rangle \\
& & \text{and for } 1 \leq i \leq n, V_i^* = \gamma(V_i, E)
\end{array}
$$

Figure 5.3 Extending the C_aEK-machine with *callcc*

direct compilers use transformations similar to the A-reductions on an *ad hoc* and incomplete basis. It is therefore natural to modify such compilers to perform a complete A-normalization phase, and analyze the effects. We have conducted such an experiment with the non-optimizing, direct compiler CAML Light [59].[15] This compiler translates ML programs into byte-code via a λ-calculus based intermediate language, and then interprets this byte-code. By performing A-normalization on the intermediate language and rewriting the interpreter as a C_aEK machine, we achieved speedups of between 50% and 100% for each of a dozen small benchmarks. Naturally, we expect the speedups to be smaller when modifying an optimizing compiler.

[15]We gratefully thank Cormac Flanagan for implementing this experiment.

Chapter 6

Correspondence of Data Flow Analyzers

The final component of compilers that we will study is data-flow analysis. The literature generally supports the idea that CPS improves the precision of data flow analysis [10, 15, 33, 34, 58, 92]. Although much of the evidence is informal, investigations by Nielson [72] and Burn/Filho [33, 34] support, to some degree, the idea with formal results.[16] However, their results do not pinpoint the source of increased abstract information and do not explain the observation of many people that continuation-passing confuses some conventional data flow analyses.

Prompted by conversations with G. Burn [July 92] and with H. Boehm [August 92], and by the observation that the CPS transformation obscures some obvious properties of programs, we started to investigate the exact effect of the CPS program representation on the data flow analysis of programs. To study the problem, we derive three canonical data flow analyzers for the core of our Scheme-like language. The first analyzer is based on a direct semantics of the language, the second on a continuation-semantics of the language, and the last on the direct semantics of CPS terms. All analyzers compute the control flow graph of the source program and hence our results apply to a large class of data flow analyses. We then formally compare the information gathered by the different analyzers.

The next section derives a data flow analyzer from the direct semantics of Core Scheme. Section 2 repeats the derivations for the CPS data flow analyzers. In Section 3, we presents all the formal theorems relating the analyses, which are discussed from a practical perspective in Section 4. The results in this chapter ap-

[16] We discussed and re-confirmed the informal idea that CPS improves program analysis with, among others, Charles Consel, and Olivier Danvy at LFP '92 [June 92], following the presentation of our paper on equational reasoning about programs in CPS [85]; in an email exchange with Geoffrey Burn and Juarez Filho [July 92]; in further discussions at POPL '93 with Daniel Weise; in email discussions with Kelsey [July 93] and Shivers [May 93]; and in discussions with Burn at FPCA '93 [June 93]. We suspect that many others who use the CPS transformation subscribe to the same conjecture.

peared in a paper presented at the Conference on Programming Language Design and Implementation [86, 87].

6.1 Constant Propagation by Abstract Interpretation

For the analysis of programs, we assume the programs to be in A-normal form, and all the bound variables to be unique, *e.g.*, if $(\lambda x_1.M_1)$ and $(\lambda x_2.M_2)$ are two distinct procedures in a program, then $x_1 \neq x_2$. The restricted subset is the following language:

$$\begin{array}{rcl} M & ::= & V \mid (\mathbf{let}\ (x\ V)\ M) \mid (\mathbf{let}\ (x\ (V\ V))\ M) \mid (\mathbf{let}\ (x\ (\mathbf{if0}\ V\ M\ M))\ M) \\ V & ::= & n \mid x \mid \mathsf{add1} \mid \mathsf{sub1} \mid (\lambda x.M) \end{array}$$

The normalization process does not affect the results of the data flow analyzers and is only for convenience. Intuitively, the first phase of A-normalization gives every subexpression a name to which the data flow analyzer can associate information about the expression. Without A-normalization, the analyzer would typically associate a "label" with every expression and attach the information about each expression at the corresponding label [53, 72, 92]. The two treatments are identical but the elimination of a separate notion of labels simplifies the analyzers. The second phase of A-normalization re-orders the expressions to reflect the order in which the interpreters will traverse them. For example, an expression $(\mathsf{add1}\ (\mathsf{let}\ (x\ V)\ 0))$ would be rewritten as $(\mathbf{let}\ (x\ V)\ (\mathsf{add1}\ 0))$.

The semantics of the restricted subset of Λ is specified by the two predicates $\mathcal{M}$ and *app* defined in Figure 6.1. It is straightforward to show by induction on the height of the trees that if $\langle M, \rho, s\rangle\ \mathcal{M}\ A_1$ and $\langle M, \rho, s\rangle\ \mathcal{M}\ A_2$, then $A_1 = A_2$. Consequently, $\mathcal{M}$ is a partial function from terms, environments, and stores to answers. The environment is a finite table that maps the free variables to locations; the store is a finite table that maps locations to values.[17] An answer is a pair that consists of a run-time value and a store. The set of run-time values consists of numbers and closures. A *closure* is one of the procedure tags **inc** and **dec**, or a data-structure that contains the text of a user-defined procedure and the environment at the point of the creation of the closure. When applying a closure $\langle \mathbf{cl}\ x, M, \rho\rangle$ to a value u, we extend the

[17]Technically, the formulation of the semantics does not require a store. However, the presence of the store is useful for the derivation of data flow analyzers, and allows us to extend the result to a more realistic language that includes assignments.

Domains:

$$\begin{array}{rcl} Ans & = & Val \times Sto \\ Env & = & Var \multimap Loc \\ Sto & = & Loc \multimap Val \\ Val & = & Num + Clo \\ Clo & = & (Var \times \Lambda \times Env) + \mathbf{inc} + \mathbf{dec} \end{array}$$

Auxiliary Function:

$$\begin{array}{rcl} \phi : \Lambda(V) \times Env \times Sto & \rightarrow & Val \\ \phi(n, \rho, s) & = & n \\ \phi(x, \rho, s) & = & s(\rho(x)) \\ \phi(\mathsf{add1}, \rho, s) & = & \mathbf{inc} \\ \phi(\mathsf{sub1}, \rho, s) & = & \mathbf{dec} \\ \phi((\lambda x.M), \rho, s) & = & \langle \mathbf{cl}\ x, M, \rho \rangle \end{array}$$

$\mathcal{M} : (\Lambda \times Env \times Sto) \rightarrow Ans$

$$\frac{u = \phi(V, \rho, s)}{\langle V, \rho, s \rangle\ \mathcal{M}\ \langle u, s \rangle} \qquad \frac{u = \phi(V, \rho, s) \quad \langle M, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \rho, s \rangle\ \mathcal{M}\ A}$$

$$\frac{\begin{array}{c} u_1 = \phi(V_1, \rho, s) \quad u_2 = \phi(V_2, \rho, s) \quad \langle u_1, u_2, s \rangle\ app\ \langle u_3, s_3 \rangle \\ \langle M, \rho[x := new(x)], s_3[new(x) := u_3] \rangle\ \mathcal{M}\ A \end{array}}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \rho, s \rangle\ \mathcal{M}\ A}$$

$$\frac{u_0 = \phi(V_0, \rho, s) \quad \langle M_i, \rho, s \rangle\ \mathcal{M}\ \langle u_1, s_1 \rangle \quad \langle M, \rho[x := new(x)], s_1[new(x) := u_1 \rangle\ \mathcal{M}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \rho, s \rangle\ \mathcal{M}\ A}$$

where $i = 1$ if $u_0 = 0$ and $i = 2$ otherwise.

$app : (Val \times Val \times Sto) \rightarrow Ans$

$$\frac{}{\langle \mathbf{inc}, n, s \rangle\ app\ \langle (n+1), s \rangle} \qquad \frac{}{\langle \mathbf{dec}, n, s \rangle\ app\ \langle (n-1), s \rangle}$$

$$\frac{\langle M, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}\ A}{\langle \langle \mathbf{cl}\ x, M, \rho \rangle, u, s \rangle\ app\ A}$$

Figure 6.1 Direct Interpreter for Core Scheme

enclosed environment at x with a *new* location and extend the store with the value u at the new location. Thus, the bound variable of a procedure or a block is related to different locations, one for each invocation of the procedure. The function *new* takes a variable x, a store s, and returns a new location ℓ from which it is possible to recover x, *i.e.*, $new(x, s) = \ell \notin dom(s)$ and $x = new^{-1}(\ell)$. (For brevity, we will usually omit the second argument to *new* since it is easy to reconstruct from the context.)

Based on well-known ideas from the area of abstract interpretation [16, 53, 72, 92], we derive a data flow analyzer from the previous interpreter. The first step in the derivation is to associate *one* location with each variable that holds the set of all values to which the variable is bound during the evaluation of the program. Second, we approximate the infinite sets of values, and modify the interpreter to detect and recover from all loops when computing over the universe of approximate values.

6.1.1 Abstracting Procedures

The first step in the derivation of the data flow analyzer is to limit the number of locations that can be created during the evaluation of a given program. One of the simple approximations, known as 0CFA analysis [82, 92], is to associate *one* location for each variable and to collect *all* the values to which the variable is bound at that location. Formally, we approximate environments ρ to $\overline{\rho}$ and stores s to $\overline{s}$ as follows:

- Since each variable is associated with exactly one location, we can choose that location to be the variable itself. Thus, if $\rho = \{x_1 \mapsto new(x_1), x_2 \mapsto new(x_2), \ldots\}$, then the abstract environment is:

  $$\begin{aligned}\overline{\rho} &= \{x_1 \mapsto new^{-1}(new(x_1)), x_2 \mapsto new^{-1}(new(x_2)), \ldots\} \\ &= \{x_1 \mapsto x_1, x_2 \mapsto x_2, \ldots\}.\end{aligned}$$

  As the approximation of the environment does not provide any information, we dispense with the environment in the remainder of the chapter. The approximation (and subsequent elimination) of environments immediately entails an approximation of closures. A closure $\langle \mathbf{cl}\ x, M, \rho \rangle$ becomes an abstract closure $\langle \mathbf{cl}^{\ominus}\ x, M \rangle$. It follows that abstract closures need not be approximated any further since, given a source program, the sets of abstract closures are finite.

- For stores $s = \{new(x_1) \mapsto u_1, new(x_2) \mapsto u_2, \ldots\}$, we first recover the variable associated with each of the locations: $\{x_1 \mapsto u_1, x_2 \mapsto u_2, \ldots\}$. Then, to obtain the store $\overline{s}$, we merge all entries of the form $x \mapsto u_1, x \mapsto u_2, \ldots$ for some x into one entry $x \mapsto \{u_1, u_2, \ldots\}$.

A "collecting semantics" like above associates a set with each variable. Intuitively, the larger the set, the less information is available at compile time about the variable. To formalize this notion of "precision", we note that the sets of collected values form a complete lattice ordered by set-inclusion and whose least upper bound is set-union. Thus, the relation "is more precise than" coincides with the lattice ordering.

6.1.2 Abstracting Integers

Despite the approximations of environments, stores, and closures, the "collecting semantics" may still associate an infinite set of values, *e.g.*, $\{0, 1, 2, \ldots\}$, with a given variable. The problem is that the lattice of collected values contains infinite chains of elements of decreasing precision, *e.g.*, $\emptyset \subseteq \{0\} \subseteq \{0, 1\} \subseteq \{0, 1, 2\} \ldots$. Since these infinite chains may cause divergence, we need to approximate sets of numbers to abstract numbers [54]:

$$\overline{\emptyset} = \bot, \quad \overline{\{n\}} = n, \quad \text{and} \quad \overline{\{n_1, n_2, \ldots\}} = \top \qquad\qquad n_1 \neq n_2.$$

At this point, the universe of abstract values consists of abstract numbers and abstract closures. It remains to impose an order $\sqsubseteq$ on the abstract values similar to the order $\subseteq$ on collected values that coincides with the relation "is more precise than". We organize the abstract values in a lattice that is the product of two lattices: the first is the traditional lattice $\mathbb{N}_\bot^\top$ used for constant propagation [54], and the second is the power set of abstract closures relative to the given program (ordered by the subset relation) used to approximate the control flow [92]. The ordering relation $\sqsubseteq$, as well as the least upper bound operation $\sqcup$, of the lattice of abstract values are defined component-wise. It is easy to check that, if S_1 and S_2 are sets of collected values, then $S_1 \subseteq S_2$ implies $\overline{S_1} \sqsubseteq \overline{S_2}$.

The ordering of abstract values induces an ordering on stores. If σ_1 and σ_2 are stores that map variables to abstract values, then $\sigma_1 \sqsubseteq \sigma_2$ if for every variable x in the domain of σ_1, we have that $\sigma_1(x) \sqsubseteq \sigma_2(x)$. The latter ordering also induces an ordering on abstract answers, *i.e.*, pairs of abstract values and abstract stores, computed component-wise.

We can now specify the collecting interpreter that manipulates abstract values (Figures 6.2 and 6.3). The interpreter uses the abstract primitive operations $\mathit{add1}^\ominus$ and $\mathit{sub1}^\ominus$ instead of the precise ones:

$$\begin{array}{rlcllcllcl} & \mathit{add1}^\ominus(\bot) & = & \bot & \mathit{add1}^\ominus(n) & = & (n+1) & \mathit{add1}^\ominus(\top) & = & \top \\ \text{and} & \mathit{sub1}^\ominus(\bot) & = & \bot & \mathit{sub1}^\ominus(n) & = & (n-1) & \mathit{sub1}^\ominus(\top) & = & \top \end{array}$$

6.1.3 Correctness

The correctness criterion of an abstract collecting interpreter is that its results approximate the actual execution of the program. For example, if the variable x gets

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\text{Domains:}}\\
\overline{Ans} & = & \overline{Val} \times \overline{Sto}\\
\overline{Sto} & = & \overline{Var} \multimap \overline{Val}\\
\overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo})\\
\overline{Clo} & = & (Var \times \Lambda) + \mathbf{inc} + \mathbf{dec}
\end{array}
\qquad
\begin{array}{rcl}
\multicolumn{3}{c}{\text{Auxiliary Functions:}}\\
\phi^{\ominus} : \Lambda(V) \to \overline{Sto} & \to & \overline{Val}\\
\phi^{\ominus}(n, \sigma) & = & \langle n, \emptyset \rangle\\
\phi^{\ominus}(x, \sigma) & = & \sigma(x)\\
\phi^{\ominus}(\mathsf{add1}, \sigma) & = & \langle \bot, \{\mathbf{inc}\} \rangle\\
\phi^{\ominus}(\mathsf{sub1}, \sigma) & = & \langle \bot, \{\mathbf{dec}\} \rangle\\
\phi^{\ominus}((\lambda x.M), \sigma) & = & \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, M \rangle\} \rangle
\end{array}
$$

$\mathcal{M}^{\ominus} : (\Lambda \times \overline{Sto}) \to \overline{Ans}$

$$
\frac{u = \phi^{\ominus}(V, \sigma)}{\langle V, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u, \sigma \rangle}
\qquad
\frac{u = \phi^{\ominus}(V, \sigma) \qquad \langle M, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ V)\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}
$$

$$
\frac{u_i = \phi^{\ominus}(V_i, \sigma) \quad \langle u_1, u_2, \sigma \rangle\ app^{\ominus}\ \langle u_3, \sigma_3 \rangle \quad \langle M, \sigma_3[x := \sigma_3(x) \sqcup u_3] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}
$$

where $i = 1, 2$

$$
\frac{u_0 = \phi^{\ominus}(V_0, \sigma) \quad \langle M_i, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_1, \sigma_1 \rangle \quad \langle M, \sigma_1[x := \sigma_1(x) \sqcup u_1] \rangle\ \mathcal{M}^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}
$$

where $i = 1$ if $u_0 = \langle 0, \emptyset \rangle$ and $i = 2$ if $\langle 0, \emptyset \rangle \not\sqsubseteq u_0$.

$$
\frac{\begin{array}{c}\langle 0, \emptyset \rangle \sqsubset u_0 = \phi^{\ominus}(V_0, \sigma) \quad \langle M_1, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_1, \sigma_1 \rangle \quad \langle M_2, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_2, \sigma_2 \rangle\\ \langle M, (\sigma_1 \sqcup \sigma_2)[x := (\sigma_1 \sqcup \sigma_2)(x) \sqcup (u_1 \sqcup u_2)] \rangle\ \mathcal{M}^{\ominus}\ A\end{array}}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \sigma \rangle\ \mathcal{M}^{\ominus}\ A}
$$

$app^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Sto}) \to \overline{Ans}$

$$
\frac{\langle cl_1, u, \sigma \rangle\ app_1{}^{\ominus}\ A_1 \quad \ldots \quad \langle cl_n, u, \sigma \rangle\ app_1{}^{\ominus}\ A_n}{\langle \langle n, \{cl_1, \ldots, cl_n\} \rangle, u, \sigma \rangle\ app^{\ominus}\ \bigsqcup_{i=1,n} A_i}
$$

Figure 6.2 Direct Abstract Collecting Interpreter (I)

bound to 5 along any actual execution path, the abstract collecting interpreter should associate an abstract value $u \sqsupseteq \langle 5, \bot \rangle$ with the variable x.

Lemma 6.1 Let $\overline{s} \sqsubseteq \sigma$, $\overline{\{u_1\}} \sqsubseteq u_2$ and $\overline{s_1} \sqsubseteq \sigma_2$. If $\langle M, \rho, s \rangle\ \mathcal{M}\ \langle u_1, s_1 \rangle$, then $\langle M, \sigma \rangle\ \mathcal{M}^{\ominus}\ \langle u_2, \sigma_2 \rangle$.

$$app_1^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Sto}) \to \overline{Ans}$$

$$\frac{u = \langle add1^{\ominus}(n), \emptyset\rangle}{\langle \mathbf{inc}, \langle n, CL\rangle, \sigma\rangle \ app_1^{\ominus} \ \langle u, \sigma\rangle} \qquad \frac{u = \langle sub1^{\ominus}(n), \emptyset\rangle}{\langle \mathbf{dec}, \langle n, CL\rangle, \sigma\rangle \ app_1^{\ominus} \ \langle u, \sigma\rangle}$$

$$\frac{\langle M, \sigma[x := \sigma(x) \sqcup u]\rangle \ \mathcal{M}^{\ominus} \ A}{\langle\langle \mathbf{cl}^{\ominus} \ x, M\rangle, u, \sigma\rangle \ app_1^{\ominus} \ A}$$

Figure 6.3 Direct Abstract Collecting Interpreter (II)

Proof Idea The proof proceeds by induction on the height of the derivation of $\langle u_1, s_1\rangle$. The base case is when M is a value V. Then $u_1 = \phi(V, \rho, s)$, $s_1 = s$, $u_2 = \phi^{\ominus}(V, \sigma)$, and $\sigma_2 = \sigma$. It suffices to check that $\overline{\{u_1\}} \sqsubseteq u_2$ which is straightforward. We only show one inductive case when $M = (\mathbf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ N)$. Let $u_0 = \phi(V_0, \rho, s)$, then as before we have that $\overline{\{u_0\}} \sqsubseteq \phi^{\ominus}(V_0, \sigma)$. There are two cases:

1. $u_0 = 0$, then $\langle 0, \emptyset\rangle \sqsubseteq \phi^{\ominus}(V_0, \sigma)$, *i.e.*, either $\langle 0, \emptyset\rangle = \phi^{\ominus}(V_0, \sigma)$ or $\langle 0, \emptyset\rangle \sqsubset \phi^{\ominus}(V_0, \sigma)$. The abstract collecting interpreter proceeds with the evaluation of M_1 or with the evaluation of both M_1 and M_2. The result follows by induction and the fact that the results of the two branches are combined.

2. $u_0 \neq 0$, then u_0 can be any number n other than 0 or a closure. It follows that $\phi^{\ominus}(V_0, \sigma) \sqsupseteq \langle n, CL\rangle$ or $\phi^{\ominus}(V_0, \sigma) \sqsupseteq \langle \top, CL\rangle$ where CL is an arbitrary set of closures. Either M_2 is evaluated or both branches are evaluated, and the result follows as in the first case.

□

6.1.4 Termination

When interpreted naïvely, the specification for the abstract collecting interpreter may be perceived as a partial function that diverges on some inputs, and hence is not quite the desired data flow analysis. This is not a problem, since it is possible to detect (and hence avoid) all loops.

More precisely, assume we have the following fragment of a derivation tree:

$$\begin{array}{l} \ldots \\ \langle M_j, \sigma_j \rangle \ \mathcal{M}^{\ominus} \ ? \\ \ldots \\ \langle M_i, \sigma_i \rangle \ \mathcal{M}^{\ominus} \ ? \\ \ldots \\ \langle M_1, \sigma_1 \rangle \ \mathcal{M}^{\ominus} \ ? \end{array}$$

The evaluation of M_1 in store σ_1 requires the value of M_i in store σ_i, which in turn requires the value of M_j in store σ_j and so on. If the above fragment of the derivation is indeed infinite, then one of the M's must be repeated infinitely often as the abstract syntax tree of the program has only a finite number of subtrees. Thus, without loss of generality, let $M_1 = M_i = M_j = M$. By inspection of the abstract collecting interpreter, we have $\sigma_1 \ldots \sqsubseteq \sigma_i \ldots \sqsubseteq \sigma_j \ldots$. As the lattice of abstract stores does not have any infinite ascending chains, one of the σ's in the sequence must be repeated, say $\sigma_i = \sigma_j = \sigma$. Thus, all loops will result in two calls to the abstract collecting interpreter with the same arguments.

By modifying the abstract collecting interpreter to compare its arguments with a history of all the previous arguments, we can therefore detect all loops. Having detected a loop, we return the least informative value paired with the current store. Thus, if the arguments $\langle M, \sigma \rangle$ have already been considered, the interpreter returns the answer $\langle\langle \top, CL^{\top} \rangle, \sigma\rangle$ where $CL^{\top}$ is the set of all abstract closures $\langle \mathbf{cl}^{\ominus}\ x, M \rangle$ in the program.

In the remainder of the chapter, we will use "abstract collecting interpreter" or "data flow analysis" to refer to the terminating versions of the interpreter that detects loops as above.[18]

6.2 CPS Analyzers

The continuation-passing style transformation may be applied to the interpreter (if we think of the interpreter as a program) or to the source program. To distinguish the two approaches we will refer to the first transformation as the *semantic*-CPS transformation and to the second as the *syntactic*-CPS transformation. We discuss both possibilities in this section.

[18]The termination argument of our data flow analyzer is not based on computing the fixed-point of a function by iteration.

6.2.1 Semantic-CPS Transformation

The CPS interpreter (see Figure 6.4) maps expressions, environments, continuations, and stores to answers. It employs two auxiliary functions: *appk*, which is the CPS counterpart of *app*, and *appr*, which corresponds to the "return" operation of an abstract machine. The latter operation binds the return value to a variable, restores the environment, pops the control stack, and jumps to the next instruction.

Domains:

$$\begin{array}{rcl@{\qquad\qquad}rcl} Ans & = & Val \times Sto & Con & = & (\Lambda(E) \times Env) :: Con + \mathbf{nil} \\ Env & = & Var \multimapdotbothvert Loc & Val & = & Num + Clo \\ Sto & = & Loc \multimapdotbothvert Val & Clo & = & (Var \times \Lambda \times Env) + \mathbf{inc} + \mathbf{dec} \end{array}$$

$\mathcal{C} : (\Lambda \times Env \times Con \times Sto) \to Ans$

$$\frac{u = \phi(V, \rho, s) \quad \langle \kappa, \langle u, s\rangle\rangle \ appr \ A}{\langle V, \rho, \kappa, s\rangle \ \mathcal{C} \ A}$$

$$\frac{u = \phi(V, \rho, s) \quad \langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C} \ A}{\langle (\mathsf{let}\ (x\ V)\ M), \rho, \kappa, s\rangle \ \mathcal{C} \ A}$$

$$\frac{u_1 = \phi(V_1, \rho, s) \quad u_2 = \phi(V_2, \rho, s) \quad \langle u_1, u_2, \langle (\mathsf{let}\ (x\ [\ ])\ M), \rho\rangle :: \kappa, s\rangle \ appk \ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \rho, \kappa, s\rangle \ \mathcal{C} \ A}$$

$$\frac{u_0 = \phi(V_0, \rho, s) \quad \langle M_i, \rho, \langle (\mathsf{let}\ (x\ [\ ])\ M), \rho\rangle :: \kappa, s\rangle \ \mathcal{C} \ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \rho, \kappa, s\rangle \ \mathcal{C} \ A} \quad i = 1 \text{ if } u_0 = 0,\ i = 2 \text{ otherwise.}$$

$appk : (Val \times Val \times Con \times Sto) \to Ans$

$$\frac{\langle \kappa, \langle (n+1), s\rangle\rangle \ appr \ A}{\langle \mathbf{inc}, n, \kappa, s\rangle \ appk \ A} \qquad\qquad \frac{\langle \kappa, \langle (n-1), s\rangle\rangle \ appr \ A}{\langle \mathbf{dec}, n, \kappa, s\rangle \ appk \ A}$$

$$\frac{\langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C} \ A}{\langle \langle \mathbf{cl}\ x, M, \rho\rangle, u, \kappa, s\rangle \ appk \ A}$$

$appr : (Con \times Ans) \to Ans$

$$\frac{\langle M, \rho[x := new(x)], \kappa, s[new(x) := u]\rangle \ \mathcal{C} \ A}{\langle \langle (\mathsf{let}\ (x\ [\ ])\ M), \rho\rangle :: \kappa, \langle u, s\rangle\rangle \ appr \ A} \qquad\qquad \frac{}{\langle \mathbf{nil}, A\rangle \ appr \ A}$$

Figure 6.4 Semantic-CPS Interpreter

The formal relationship between the direct and semantic-CPS interpreters is subject of the following lemma.

Lemma 6.2 Let $M \in \Lambda$.
Then $\langle M, \rho, s \rangle \ \mathcal{M} \ A$ if and only if $\langle M, \rho, \mathbf{nil}, s \rangle \ \mathcal{C} \ A$.

Proof Idea By induction on the height of the derivation of A. The statement of the lemma must be strengthened to take continuations other than **nil** into account. □

6.2.2 Syntactic-CPS Transformation

The output of the CPS transformation is the language $cps(\Lambda)$:

$$\begin{array}{lcl} P & ::= & (k\ W) \mid (\mathbf{let}\ (x\ W)\ P) \mid (W\ W\ (\lambda x.P)) \mid (\mathbf{let}\ (k\ \lambda x.P)\ (\mathbf{if0}\ W\ P\ P)) \\ W & ::= & n \mid x \mid \mathtt{add1k} \mid \mathtt{sub1k} \mid (\lambda xk.P) \end{array}$$

The semantics of CPS programs is defined by $\mathcal{M}_c$, a specialized version of the direct interpreter $\mathcal{M}$ [38]. The evaluator for CPS terms (see Figure 6.5) handles procedures of two arguments and manipulates a larger set of run-time values than the direct interpreter that includes continuations of the form $\langle \mathbf{co}\ x, P, \rho \rangle$. The enlarged set of run-time values reflects the salient aspect of the CPS transformation: it reifies the continuation of the evaluator to an object that the program text may manipulate.[19]

In order to establish the formal relationship between the semantics of a direct term and the semantics of its CPS-transform, we define the function δ that relates direct run-time values to their CPS counterparts:

$$\delta(n) = n,\ \delta(\mathbf{inc}) = \mathbf{inck},\ \delta(\mathbf{dec}) = \mathbf{deck},\ \delta(\langle \mathbf{cl}\ x, M, \rho \rangle) = \langle \mathbf{cl}\ xk, \mathcal{F}_k[\![M]\!], \rho \rangle$$

We extend δ to work on stores by applying it to the value at each location and to answers by applying it to both the value component and the store component.

The following lemma gives the precise connection between the semantic-CPS interpreter and the syntactic-CPS interpreter. The lemma states that the interpreters yield answers related by δ with the understanding that the store resulting from the

[19] In theory, we could use the direct interpreter $\mathcal{M}$ to evaluate CPS programs. However, this choice forces continuations to be represented as procedures, which is unrealistic and confusing for data flow analyzers.

Domains:

$$
\begin{array}{rcl}
Ans & = & Val \times Sto \\
Env & = & Var \multimap Loc \\
Sto & = & Loc \multimap Val \\
Val & = & Num + Clo + Con \\
Clo & = & (Var \times KVar \times cps(\Lambda) \times Env) \\
 & & + \,\mathbf{inck} + \mathbf{deck} \\
Con & = & (Var \times cps(\Lambda) \times Env) + \mathbf{stop}
\end{array}
$$

Auxiliary Function:

$$
\begin{array}{rcl}
\phi_c : cps(\Lambda)(W) \times Env \times Sto & \rightarrow & Val \\
\phi_c(n, \rho, s) & = & n \\
\phi_c(x, \rho, s) & = & s(\rho(x)) \\
\phi_c(\mathsf{add1k}, \rho, s) & = & \mathbf{inck} \\
\phi_c(\mathsf{sub1k}, \rho, s) & = & \mathbf{deck} \\
\phi_c((\lambda xk.P), \rho, s) & = & \langle \mathbf{cl}\ xk, P, \rho \rangle
\end{array}
$$

$\mathcal{M}_c : (cps(\Lambda) \times Env \times Sto) \rightarrow Ans$

$$
\frac{\kappa = s(\rho(k)) \quad u = \phi_c(W, \rho, s) \quad \langle \kappa, \langle u, s \rangle \rangle \ appr_c\ A}{\langle (k\ W), \rho, s \rangle\ \mathcal{M}_c\ A}
$$

$$
\frac{u = \phi_c(W, \rho, s) \quad \langle P, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}_c\ A}{\langle (\mathsf{let}\ (x\ W)\ P), \rho, s \rangle\ \mathcal{M}_c\ A}
$$

$$
\frac{u_1 = \phi_c(W_1, \rho, s) \quad u_2 = \phi_c(W_2, \rho, s) \quad \langle u_1, u_2, \langle \mathbf{co}\ x, P, \rho \rangle, s \rangle\ app_c\ A}{\langle (W_1\ W_2\ (\lambda x.P)), \rho, s \rangle\ \mathcal{M}_c\ A}
$$

$$
\frac{u_0 = \phi_c(W_0, \rho, s) \quad \langle P_i, \rho[k := new(k)], s[new(k) := \langle \mathbf{co}\ x, P, \rho \rangle] \rangle\ \mathcal{M}_c\ A}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \rho, s \rangle\ \mathcal{M}_c\ A}
$$

where $i = 1$ if $u_0 = 0$ and $i = 2$ otherwise.

$app_c : (Val \times Val \times Val \times Sto) \rightarrow Ans$

$$
\frac{\langle \kappa, \langle (n+1), s \rangle \rangle\ appr_c\ A}{\langle \mathbf{inck}, n, \kappa, s \rangle\ app_c\ A} \qquad \frac{\langle \kappa, \langle (n-1), s \rangle \rangle\ appr_c\ A}{\langle \mathbf{deck}, n, \kappa, s \rangle\ app_c\ A}
$$

$$
\frac{\langle P, \rho[x := new(x), k := new(k)], s[new(x) := u, new(k) := \kappa] \rangle\ \mathcal{M}_c\ A}{\langle \langle \mathbf{cl}\ xk, P, \rho \rangle, u, \kappa, s \rangle\ app_c\ A}
$$

$appr_c : (Val \times Ans) \rightarrow Ans$

$$
\frac{\langle P, \rho[x := new(x)], s[new(x) := u] \rangle\ \mathcal{M}_c\ A}{\langle \langle \mathbf{co}\ x, P, \rho \rangle, \langle u, s \rangle \rangle\ appr_c\ A} \qquad \frac{}{\langle \mathbf{stop}, A \rangle\ appr_c\ A}
$$

Figure 6.5 Syntactic-CPS Interpreter

syntactic-CPS interpreter will contain additional entries that correspond to continuations. Given Lemma 6.2, the lemma also relates the syntactic-CPS interpreter to the direct one.

Lemma 6.3 Let $M \in \Lambda$, then:

$$\begin{array}{l}\langle M, \rho, \mathbf{nil}, s\rangle\ \mathcal{C}\ \langle u_1, s_1\rangle \text{if and only if} \\ \quad \langle \mathcal{F}_k[\![M]\!], \rho[k := new(k)], \delta(s)[new(k) := \mathbf{stop}]\rangle\ \mathcal{M}_c \\ \qquad \langle \delta(u_1), \delta(s_1)[new(k_1) := \kappa_1, new(k_2) := \kappa_2, \ldots]\rangle.\end{array}$$

Proof The proof requires a more general statement of the lemma. Let $M \in \Lambda$, and let:

$$\begin{array}{rcl} E_i & = & (\mathbf{let}\ (u_i\ [\ ])\ N_i) \\ \kappa(h, i, \ldots, j, \top) & = & \langle E_h, \rho_h\rangle :: \langle E_i, \rho_i\rangle :: \ldots :: \langle E_j, \rho_j\rangle :: \mathbf{nil} \\ \mathbf{co}(i, j) & = & \langle \mathbf{co}\ u_i, \mathcal{F}_{k_j}[\![N_i]\!], \rho_i[k_j := new(k_j)]\rangle \\ s(h, i, \ldots, j, \top) & = & s[new(k_h) := \mathbf{co}(h, i), \ldots, new(k_j) := \mathbf{co}(j, \top), \\ & & \quad new(k_\top) := \mathbf{stop}]\rangle \end{array}$$

then:

$$\begin{array}{l}\langle M, \rho, \kappa(h, \ldots, \top), s\rangle\ \mathcal{C}\ \langle u_1, s_1\rangle \quad \text{if and only if} \\ \quad \langle \mathcal{F}_{k_h}[\![M]\!], \rho[k_h := new(k_h)], \delta(s)(q, \ldots, h, \ldots, \top)\rangle\ \mathcal{M}_c \\ \qquad \langle \delta u_1, \delta(s_1)(p, \ldots, q, \ldots, h, \ldots, \top)\rangle\end{array}$$

The proof is by induction on the height of the derivation of $\langle u_1, s_1\rangle$. For the base case (height $= 1$), we must have that M is a value V and the continuation is the initial one:

$$\frac{u = \phi(V, \rho, s) \quad \langle \mathbf{nil}, \langle u, s\rangle\rangle\ appr\ \langle u, s\rangle}{\langle V, \rho, \kappa(\top), s\rangle\ \mathcal{C}\ \langle u, s\rangle}$$

$$\frac{\kappa = \mathbf{stop} \quad u' = \phi_c(\mathcal{V}[\![V]\!], \rho[k_\top := new(k_\top)], s') \quad \langle \mathbf{stop}, \langle u', s'\rangle\rangle\ appr_c\ \langle u', s'\rangle}{\langle (k_\top\ \mathcal{V}[\![V]\!]), \rho[k_\top := new(k_\top)], s'\rangle\ \mathcal{M}_c\ \langle u', s'\rangle}$$

where $s' = \delta(s)(q, \ldots, \top)$. The result follows since $u' = \delta(u)$.

For the inductive case (height > 1), we proceed by cases of M:

- $M = V$:

$$\frac{u = \phi(V, \rho, s) \quad \dfrac{\langle N_h, \rho_h[u_h := new(u_h)], \kappa(i, \ldots, \top), s[new(u_h) := u]\rangle\ \mathcal{C}\ A}{\langle \kappa(h, i, \ldots, \top), \langle u, s\rangle\rangle\ appr\ A}}{\langle V, \rho, \kappa(h, i, \ldots, \top), s\rangle\ \mathcal{C}\ A}$$

$$\frac{\dfrac{\langle \mathcal{F}_{k_i}[\![N_h]\!], \rho_h[u_h := new(u_h)][k_i := new(k_i)], s'[new(u_h) := u']\rangle\ \mathcal{M}_c\ A'}{\langle \mathbf{co}(h, i), \langle u', s'\rangle\rangle\ appr_c\ A'}}{\langle (k_h\ \mathcal{V}[\![V]\!]), \rho[k_h := new(k_h)], s'\rangle\ \mathcal{M}_c\ A'}$$

where $u' = \delta(u)$ and $s' = \delta(s)(q, \ldots, h, i, \ldots, \top)$. The result follows since A and A' are correctly related by induction.

- $M = (\textbf{let}\ (x\ V)\ N)$:

$$\frac{u = \phi(V, \rho, s) \quad \langle N, \rho[x := new(x)], \kappa(h, \ldots, \top), s[new(x) := u] \rangle\ \mathcal{C}\ A}{\langle (\textbf{let}\ (x\ V)\ N), \rho, \kappa(h, \ldots, \top), s \rangle\ \mathcal{C}\ A}$$

$$\frac{u' = \delta(u) \quad \langle \mathcal{F}_{k_h}[\![N]\!], \rho[k_h := new(k_h)][x := new(x)], s'[new(x) := u'] \rangle\ \mathcal{M}_c\ A'}{\langle (\textbf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_{k_h}[\![N]\!]), \rho[k_h := new(k_h)], s' \rangle\ \mathcal{M}_c\ A'}$$

where $s' = \delta(s)(q, \ldots, h, \ldots, \top)$. The result follows since A and A' are correctly related by induction.

- $M = (\textbf{let}\ (u_g\ (V_1\ V_2))\ N_g)$:

$$\frac{u_1 = \phi(V_1, \rho_g, s) \quad u_2 = \phi(V_2, \rho_g, s) \quad \langle u_1, u_2, \kappa(g, h, \ldots, \top), s \rangle\ appk\ A}{\langle (\textbf{let}\ (u_g\ (V_1\ V_2))\ N_g), \rho_g, \kappa(h, \ldots, \top), s \rangle\ \mathcal{C}\ A}$$

$$\frac{u_1' = \delta(u_1) \quad u_2' = \delta(u_2) \quad \langle u_1', u_2', \textbf{co}(g, h), s' \rangle\ app_c\ A'}{\langle (\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])), \rho_g[k_h := new(k_h)], s' \rangle\ \mathcal{M}_c\ A'}$$

where $s' = \delta(s)(q, \ldots, h, \ldots, \top)$. We proceed by cases of u_1:

  - $u_1 = \textbf{inc}$ and $u_2 = n$:

$$\frac{\dfrac{\langle N_g, \rho_g[u_g := new(u_g)], \kappa(h, \ldots, \top), s[new(u_g) := n'] \rangle\ \mathcal{C}\ A}{\langle \kappa(g, h, \ldots, \top), \langle n', s \rangle \rangle\ appr\ A}}{\langle \textbf{inc}, n, \kappa(g, h, \ldots, \top), s \rangle\ appk\ A}$$

$$\frac{\dfrac{\langle \mathcal{F}_{k_h}[\![N_g]\!], \rho_g[k_h := new(k_h)][u_g := new(u_g)], s'[new(u_g) := n'] \rangle\ \mathcal{M}_c\ A'}{\langle \textbf{co}(g, h), \langle n', s' \rangle \rangle\ appr_c\ A'}}{\langle \textbf{inck}, n, \textbf{co}(g, h), s' \rangle\ app_c\ A'}$$

  where $n' = (n + 1)$. The result follows since A and A' are correctly related by induction.

  - $u_1 = \textbf{dec}$: similar to the previous case.
  - $u_1 = \langle \textbf{cl}\ z, N_z, \rho_z \rangle$:

$$\frac{\langle N_z, \rho_z[z := new(z)], \kappa(g, h, \ldots, \top), s[new(z) := u_2] \rangle\ \mathcal{C}\ A}{\langle \langle \textbf{cl}\ z, N_z, \rho_z \rangle, u_2, \kappa(g, h, \ldots, \top), s \rangle\ appk\ A}$$

$$\frac{\langle \mathcal{F}_{k_g}[\![N_z]\!], \rho_z[z := new(z)][k_g := new(k_g)], s''[new(z) := \delta(u_2)] \rangle\ \mathcal{M}_c\ A'}{\langle \langle \textbf{cl}\ zk_g, \mathcal{F}_{k_g}[\![N_z]\!], \rho_z \rangle, \delta(u_2), \textbf{co}(g, h), s' \rangle\ app_c\ A'}$$

where $s'' = s'[new(k_g) := \mathbf{co}(g,h)]$. The result follows since A and A' are correctly related by induction.

- $M = (\mathbf{let}\ (u_g\ (\mathbf{if0}\ V_0\ M_1\ M_2))\ N_g)$:

$$\frac{u_0 = \phi(V_0, \rho_g, s) = 0 \quad \langle M_1, \rho_g, \kappa(g, h, \ldots, \top), s\rangle\ \mathcal{C}\ A}{\langle(\mathbf{let}\ (u_g\ (\mathbf{if0}\ V_0\ M_1\ M_2))\ N_g), \rho_g, \kappa(h, \ldots, \top), s\rangle\ \mathcal{C}\ A}$$

$$\frac{u_0' = \delta(u_0) \quad \langle \mathcal{F}_{k_g}[\![M_1]\!], \rho_g[k_h := new(k_h)][k_g := new(k_g)], s''\rangle\ \mathcal{M}_c\ A'}{\langle B, \rho_g[k_h := new(k_h)], s'\rangle\ \mathcal{M}_c\ A'}$$

where the term $B = (\mathbf{let}\ k_g\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])\ (\mathbf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k_g}[\![M_1]\!]\ \mathcal{F}_{k_g}[\![M_2]\!]))$, the store $s' = \delta(s)(q, \ldots, h, \ldots, \top)$ and the store $s'' = s'[new(k_g) := \mathbf{co}(g,h)]$. The result follows since A and A' are correctly related by induction. The case when the test is false is similar.

$\square$

6.2.3 Abstract Collecting Interpreters

The derivations of the abstract collecting interpreters follow the same pattern as the previous section. First, when abstracting environments, a continuation $\langle E_1, \rho_1\rangle :: \langle E_2, \rho_2\rangle :: \ldots :: \mathbf{nil}$ becomes $E_1 :: E_2 :: \ldots :: \mathbf{nil}$, and a continuation $\langle \mathbf{co}\ x, M, \rho\rangle$ becomes an abstract continuation $\langle \mathbf{co}^{\ominus}\ x, M\rangle$

The lattice of abstract values for the semantic CPS interpreter is identical to the lattice for the direct interpreter. For the syntactic-CPS collecting interpreter, the sets of values include abstract continuations and we use a lattice of abstract values that consists of the product of three lattices: the constant propagation lattice, the power set of abstract closures, and the power set of abstract continuations.

The semantic-CPS abstract collecting interpreter is in Figures 6.6 and 6.7. The syntactic-CPS abstract collecting interpreter is in Figures 6.8 and 6.9 The correctness and termination argument for both CPS interpreters is similar to the case for the direct interpreter.

Lemma 6.4 Let $\overline{s} \sqsubseteq \sigma$, $\overline{\{u_1\}} \sqsubseteq u_2$ and $\overline{s_1} \sqsubseteq \sigma_2$.

1. If $\langle M, \rho, \kappa, s\rangle\ \mathcal{C}\ \langle u_1, s_1\rangle$, then $\langle M, \overline{\kappa}, \sigma\rangle\ \mathcal{C}^{\ominus}\ \langle u_2, \sigma_2\rangle$.
2. If $\langle P, \rho, s\rangle\ \mathcal{M}_c\ \langle u_1, s_1\rangle$, then $\langle P, \sigma\rangle\ \mathcal{M}_c^{\ominus}\ \langle u_2, \sigma_2\rangle$.

Domains:

$$\begin{array}{rcl} \overline{Ans} & = & \overline{Val} \times \overline{Sto} \\ \overline{Sto} & = & Var \multimap \overline{Val} \end{array} \qquad \begin{array}{rcl} \overline{Con} & = & \Lambda(E) :: \overline{Con} + \mathbf{nil} \\ \overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo}) \\ \overline{Clo} & = & (Var \times \Lambda) + \mathbf{inc} + \mathbf{dec} \end{array}$$

$\mathcal{C}^{\ominus} : (\Lambda \times \overline{Con} \times \overline{Sto}) \to \overline{Ans}$

$$\frac{u = \phi^{\ominus}(V, \sigma) \quad \langle \kappa, \langle u, \sigma \rangle \rangle \ appr^{\ominus} \ A}{\langle V, \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A} \qquad \frac{u = \phi^{\ominus}(V, \sigma) \quad \langle M, \kappa, \sigma[x := \sigma(x) \sqcup u] \rangle \ \mathcal{C}^{\ominus} \ A}{\langle (\mathsf{let}\ (x\ V)\ M), \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A}$$

$$\frac{u_1 = \phi^{\ominus}(V_1, \sigma) \quad u_2 = \phi^{\ominus}(V_2, \sigma) \quad \langle u_1, u_2, (\mathsf{let}\ (x\ [\ ])\ M) :: \kappa, \sigma \rangle \ appk^{\ominus} \ A}{\langle (\mathsf{let}\ (x\ (V_1\ V_2))\ M), \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A}$$

$$\frac{u_0 = \phi^{\ominus}(V_0, \sigma) \quad \langle M_i, (\mathsf{let}\ (x\ [\ ])\ M) :: \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A}$$

where $i = 1$ if $u_0 = \langle 0, \emptyset \rangle$ and $i = 2$ if $\langle 0, \emptyset \rangle \not\sqsubseteq u_0$.

$$\frac{\begin{array}{c}\langle 0, \emptyset \rangle \sqsubset u_0 = \phi^{\ominus}(V_0, \sigma) \\ \langle M_1, (\mathsf{let}\ (x\ [\ ])\ M) :: \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A_1 \quad \langle M_2, (\mathsf{let}\ (x\ [\ ])\ M) :: \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A_2\end{array}}{\langle (\mathsf{let}\ (x\ (\mathsf{if0}\ V_0\ M_1\ M_2))\ M), \kappa, \sigma \rangle \ \mathcal{C}^{\ominus} \ A_1 \sqcup A_2}$$

$appk^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Con} \times \overline{Sto}) \to \overline{Ans}$

$$\frac{\langle cl_1, u, \kappa, \sigma \rangle \ appk_1{}^{\ominus} \ A_1 \quad \ldots \quad \langle cl_n, u, \kappa, \sigma \rangle \ appk_1{}^{\ominus} \ A_n}{\langle \langle n, \{cl_1, \ldots, cl_n\} \rangle, u, \kappa, \sigma \rangle \ appk^{\ominus} \ \bigsqcup_{i=1,n} A_i}$$

$appk_1{}^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Con} \times \overline{Sto}) \to \overline{Ans}$

$$\frac{u = \langle add1^{\ominus}(n), \emptyset \rangle \quad \langle \kappa, \langle u, \sigma \rangle \rangle \ appr^{\ominus} \ A}{\langle \mathbf{inc}, \langle n, CL \rangle, \kappa, \sigma \rangle \ appk_1{}^{\ominus} \ A} \qquad \frac{u = \langle sub1^{\ominus}(n), \emptyset \rangle \quad \langle \kappa, \langle u, \sigma \rangle \rangle \ appr^{\ominus} \ A}{\langle \mathbf{dec}, \langle n, CL \rangle, \kappa, \sigma \rangle \ appk_1{}^{\ominus} \ A}$$

$$\frac{\langle M, \kappa, \sigma[x := \sigma(x) \sqcup u] \rangle \ \mathcal{C}^{\ominus} \ A}{\langle \langle \mathbf{cl}^{\ominus}\ x, M \rangle, u, \kappa, \sigma \rangle \ appk_1{}^{\ominus} \ A}$$

Figure 6.6 Semantic-CPS Abstract Collecting Interpreter (I)

6.3 Formal Relationships

Given the direct and CPS-based data flow analyzers, we turn our attention to the connection between the analyses. The proofs follow the same pattern of the proofs

$appr^{\ominus} : (\overline{Con} \times \overline{Ans}) \rightarrow \overline{Ans}$

$$\frac{\langle M, \kappa, \sigma[x := \sigma(x) \sqcup u] \rangle \; \mathcal{C}^{\ominus} \; A}{\langle (\mathsf{let}\ (x\ [\ ])\ M) :: \kappa, \langle u, \sigma \rangle \rangle \; appr^{\ominus} \; A} \qquad \frac{}{\langle \mathbf{nil}, A \rangle \; appr^{\ominus} \; A}$$

Figure 6.7 Semantic-CPS Abstract Collecting Interpreter (II)

for the standard interpreters (Lemmas 6.2 and 6.3). For the connection between the direct and syntactic-CPS analyses, we need an abstract version of the function δ that maps abstract direct values to abstract CPS values:

$$\begin{aligned}
\delta^{\ominus}(\langle n, \{cl_1, \ldots, cl_i\}\rangle) &= \langle n, \{\mathcal{V}^{\ominus}(cl_1), \ldots, \mathcal{V}^{\ominus}(cl_i)\}, \emptyset\rangle \\
\mathcal{V}^{\ominus}(\langle \mathbf{cl}^{\ominus}\ x_1, M_1\rangle) &= \langle \mathbf{cl}^{\ominus}\ x_1 k_1, \mathcal{F}_{k_1}[\![M_1]\!]\rangle \\
\mathcal{V}^{\ominus}(\mathbf{inc}) &= \mathbf{inck} \\
\mathcal{V}^{\ominus}(\mathbf{dec}) &= \mathbf{deck}
\end{aligned}$$

We also extend $\delta^{\ominus}$ to work on stores by applying it to the value at each location and to answers by applying it to both components.

6.3.1 Direct versus Syntactic-CPS

The first theorem establishes that the direct analysis of M may be *more* precise than the analysis of $\mathcal{F}_k[\![M]\!]$.

Theorem 6.5 There exists a term $M \in \Lambda$ such that:

- $\langle M, \sigma \rangle \; \mathcal{M}^{\ominus} \; \langle u_1, \sigma_1 \rangle$,
- $\langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle] \rangle \; \mathcal{M}_c^{\ominus} \; \langle u_2, \sigma_2 \rangle$,
- $\delta^{\ominus}(u_1) \sqsubseteq u_2$, and for each variable in the domain of σ_1, $\delta^{\ominus}(\sigma_1(x)) \sqsubseteq \sigma_2(x)$.

Proof Let M be $(\mathbf{let}\ (a_1\ (f\ 1))\ (\mathsf{let}\ (a_2\ (f\ 2))\ a_2))$, and let:

$$\sigma = \{a_1 \mapsto \langle \bot, \emptyset \rangle,\ a_2 \mapsto \langle \bot, \emptyset \rangle,\ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, x\rangle\}\rangle,\ x \mapsto \langle \bot, \emptyset \rangle\}.$$

$$\begin{array}{rcl} & \text{Domains:} & \\ \overline{Ans} & = & \overline{Val} \times \overline{Sto} \\ \overline{Sto} & = & Var \multimap \overline{Val} \\ \overline{Val} & = & \overline{Num} \times \mathcal{P}(\overline{Clo}) \times \mathcal{P}(\overline{Con}) \\ \overline{Clo} & = & (Var \times KVar \times cps(\Lambda)) \\ & & + \ \mathbf{inck} + \mathbf{deck} \\ \overline{Con} & = & (Var \times cps(\Lambda)) + \mathbf{stop} \end{array}$$

$$\begin{array}{rcl} & \text{Auxiliary Function:} & \\ \phi_c^{\ominus} : cps(\Lambda)(W) \times \overline{Sto} & \to & \overline{Val} \\ \phi_c^{\ominus}(n, \sigma) & = & \langle n, \emptyset, \emptyset \rangle \\ \phi_c^{\ominus}(x, \sigma) & = & \sigma(x) \\ \phi_c^{\ominus}(\mathsf{add1k}, \sigma) & = & \langle \bot, \{\mathbf{inck}\}, \emptyset \rangle \\ \phi_c^{\ominus}(\mathsf{sub1k}, \sigma) & = & \langle \bot, \{\mathbf{deck}\}, \emptyset \rangle \\ \phi_c^{\ominus}((\lambda xk.P), \sigma) & = & \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ xk, P \rangle\}, \emptyset \rangle \end{array}$$

$\mathcal{M}_c^{\ominus} : (cps(\Lambda) \times \overline{Sto}) \to \overline{Ans}$

$$\frac{\kappa = \sigma(k) \quad u = \phi_c^{\ominus}(W, \sigma) \quad \langle \kappa, \langle u, \sigma \rangle \rangle \ appr_c^{\ominus}\ A}{\langle (k\ W), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

$$\frac{u = \phi_c^{\ominus}(W, \sigma) \quad \langle P, \sigma[x := \sigma(x) \sqcup u] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle (\mathsf{let}\ (x\ W)\ P), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

$$\frac{u_1 = \phi_c^{\ominus}(W_1, \sigma) \quad u_2 = \phi_c^{\ominus}(W_2, \sigma) \quad \langle u_1, u_2, \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle, \sigma \rangle\ app_c^{\ominus}\ A}{\langle (W_1\ W_2\ (\lambda x.P)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

$$\frac{u_0 = \phi_c^{\ominus}(W_0, \sigma) \quad \langle P_i, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A}$$

where $i = 1$ if $u_0 = \langle 0, \emptyset, \emptyset \rangle$ and $i = 2$ if $\langle 0, \emptyset, \emptyset \rangle \not\sqsubseteq u_0$.

$$\frac{\begin{array}{c} \langle 0, \emptyset, \emptyset \rangle \sqsubset u_0 = \phi_c^{\ominus}(W_0, \sigma) \quad \langle P_1, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A_1 \\ \langle P_2, \sigma[k := \sigma(k) \sqcup \langle \bot, \emptyset, \langle \mathbf{co}^{\ominus}\ x, P \rangle \rangle] \rangle\ \mathcal{M}_c^{\ominus}\ A_2 \end{array}}{\langle (\mathsf{let}\ (k\ \lambda x.P)\ (\mathsf{if0}\ W_0\ P_1\ P_2)), \sigma \rangle\ \mathcal{M}_c^{\ominus}\ A_1 \sqcup A_2}$$

Figure 6.8 Syntactic-CPS Abstract Collecting Interpreter (I)

It is straightforward to calculate that the result of the direct abstract collecting interpreter is $A_1 = \langle u_1, \sigma_1 \rangle$ where:

$$\begin{array}{rcl} u_1 & = & \langle \top, \emptyset \rangle \\ \sigma_1 & = & \{a_1 \mapsto \langle 1, \emptyset \rangle,\ a_2 \mapsto \langle \top, \emptyset \rangle,\ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ x, x \rangle\} \rangle,\ x \mapsto \langle \top, \emptyset \rangle\}. \end{array}$$

$$app_c^{\ominus} : (\overline{Val} \times \overline{Val} \times \overline{Val} \times \overline{Sto}) \to \overline{Ans}$$

$$\frac{\langle cl_1, u, \kappa, \sigma\rangle \; app_{1c}^{\ominus} \; A_1 \quad \ldots \quad \langle cl_n, u, \kappa, \sigma\rangle \; app_{1c}^{\ominus} \; A_n}{\langle\langle n, \{cl_1, \ldots, cl_n\}, K\rangle, u, \kappa, \sigma\rangle \; app_c^{\ominus} \; \bigsqcup_{i=1,n} A_i}$$

$$app_{1c}^{\ominus} : (\overline{Clo} \times \overline{Val} \times \overline{Val} \times \overline{Sto}) \to \overline{Ans}$$

$$\frac{u = \langle add1^{\ominus}(n), \emptyset, \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \; appr_c^{\ominus} \; A}{\langle \mathbf{inck}, \langle n, CL, K\rangle, \kappa, \sigma\rangle \; app_{1c}^{\ominus} \; A}$$

$$\frac{u = \langle sub1^{\ominus}(n), \emptyset, \emptyset\rangle \quad \langle \kappa, \langle u, \sigma\rangle\rangle \; appr_c^{\ominus} \; A}{\langle \mathbf{deck}, \langle n, CL, K\rangle, \kappa, \sigma\rangle \; app_{1c}^{\ominus} \; A}$$

$$\frac{\langle P, \sigma[x := \sigma(x) \sqcup u, k := \sigma(k) \sqcup \kappa]\rangle \; \mathcal{M}_c^{\ominus} \; A}{\langle\langle \mathbf{cl}^{\ominus} \; xk, P\rangle, u, \kappa, \sigma\rangle \; app_{1c}^{\ominus} \; A}$$

$$appr_c^{\ominus} : (\overline{Val} \times \overline{Ans}) \to \overline{Ans}$$

$$\frac{\langle \kappa_1, \langle u, \sigma\rangle\rangle \; appr_{1c}^{\ominus} \; A_1 \quad \ldots \quad \langle \kappa_n, \langle u, \sigma\rangle\rangle \; appr_{1c}^{\ominus} \; A_n}{\langle\langle n, CL, \{\kappa_1, \ldots, \kappa_n\}\rangle, \langle u, \sigma\rangle\rangle \; appr_c^{\ominus} \; \bigsqcup_{i=1,n} A_i}$$

$$appr_{1c}^{\ominus} : (\overline{Con} \times \overline{Ans}) \to Ans$$

$$\frac{\langle P, \sigma[x := \sigma(x) \sqcup u]\rangle \; \mathcal{M}_c^{\ominus} \; A}{\langle\langle \mathbf{co}^{\ominus} \; x, P\rangle, \langle u, \sigma\rangle\rangle \; appr_{1c}^{\ominus} \; A} \qquad \frac{}{\langle \mathbf{stop}, A\rangle \; appr_{1c}^{\ominus} \; A}$$

Figure 6.9 Syntactic-CPS Abstract Collecting Interpreter (II)

For the analysis of the CPS version, we have that:

$$\begin{aligned}
\mathcal{F}_k[\![M]\!] &= (f\ 1\ (\lambda a_1.(f\ 2\ (\lambda a_2.(k\ a_2))))) \\
\sigma' &= \{a_1 \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\
&\qquad a_2 \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\
&\qquad f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ xk_1, (k_1\ x)\rangle\}, \emptyset\rangle, \\
&\qquad x \mapsto \langle \bot, \emptyset, \emptyset\rangle, \\
&\qquad k \mapsto \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle\}.
\end{aligned}$$

The syntactic-CPS abstract collecting interpreter produces the answer $A_2 = \langle u_2, \sigma_2 \rangle$ where:

$$\begin{aligned}
u_2 &= \langle \top, CL^\top, K^\top \rangle \\
\sigma_2 &= \{a_1 \mapsto \langle \top, \emptyset, \emptyset \rangle, \\
&\quad a_2 \mapsto \langle \top, \emptyset, \emptyset \rangle, \\
&\quad f \mapsto \langle \bot, \{\langle \mathbf{cl}^\ominus\ xk_1, (k_1\ x) \rangle\}, \emptyset \rangle, \\
&\quad x \mapsto \langle \top, \emptyset, \emptyset \rangle, \\
&\quad k_1 \mapsto \langle \bot, \emptyset, \{\ \langle \mathbf{co}^\ominus\ a_1, (f\ 2\ (\lambda a_2.(k\ a_2))) \rangle, \langle \mathbf{co}^\ominus\ a_2, (k\ a_2) \rangle \} \rangle \\
&\quad k \mapsto \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle]
\end{aligned}$$

The analysis of the source program is more precise since it determines that the variable a_1 is constant (=1), while the analysis of the CPS program fails to produce any information about a_1. □

Our second theorem states that the direct analysis of a program may also give *less* information than the syntactic-CPS analysis. Together with the previous theorem, the result establishes that the direct analysis of a source program is *incomparable* to the syntactic-CPS analysis.

Theorem 6.6 There exists a term $M \in \Lambda$ such that:

- $\langle M, \sigma \rangle\ \mathcal{M}^\ominus\ \langle u_1, \sigma_1 \rangle$,
- $\langle \mathcal{F}_k[\![M]\!], \delta^\ominus(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\} \rangle] \rangle\ \mathcal{M}_c^\ominus\ \langle u_2, \sigma_2 \rangle$,
- $\delta^\ominus(u_1) \sqsupseteq u_2$, and for each variable x in the domain of σ_1, $\delta^\ominus(\sigma_1(x)) \sqsupseteq \sigma_2(x)$.

Proof To illustrate the principle, we present two cases in which the analysis of the CPS program yields more information than the direct analysis of the source program.

For the first case, take:

$$\begin{aligned}
M &= (\mathbf{let}\ (a_1\ (\mathbf{if0}\ x\ 0\ 1))\ (\mathbf{let}\ (a_2\ (\mathbf{if0}\ a_1\ (+\ a_1\ 3)\ (+\ a_1\ 2)))\ \ a_2)) \\
\sigma &= \{a_1 \mapsto \langle \bot, \emptyset \rangle,\ a_2 \mapsto \langle \bot, \emptyset \rangle,\ x \mapsto \langle \top, \emptyset \rangle\}
\end{aligned}$$

where $(+\ a_1\ 3)$ and $(+\ a_1\ 2)$ are the obvious abbreviations. The direct analysis of the term, not knowing which branch to take, merges the abstract values of 0 and 1 at the variable a_1 and hence loses all information about a_2. In contrast, the analysis of $\mathcal{F}_k[\![M]\!]$:

$$\mathcal{F}_k[\![M]\!] = (\mathbf{let}\ (k'\ \lambda a_1.\mathcal{F}_k[\![(\mathbf{let}\ (a_2\ (\mathbf{if0}\ a_1\ (a_1+3)\ (a_1+2)))\ a_2)]\!])\ (\mathbf{if0}\ x\ (k'\ 0)\ (k'\ 1)))$$

analyzes both $(k'\ 0)$ and $(k'\ 1)$ in a store that maps $k' \mapsto \langle \mathbf{co}^{\ominus}\ a_1, \ldots \rangle$. The analysis of each execution path determines that the abstract value of a_2 is $\langle 3, \emptyset, \emptyset \rangle$, which improves on the direct analysis.

For the second case, take M:

$$\begin{aligned} M &= (\mathbf{let}\ (a_1\ (f\ 3))\ (\mathbf{let}\ (a_2\ (\mathbf{if0}\ a_1\ 5\ (\mathsf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6)))\ a_2)) \\ \sigma &= \{a_1 \mapsto \langle \bot, \emptyset \rangle \\ &\quad\ a_2 \mapsto \langle \bot, \emptyset \rangle, \\ &\quad\ f \mapsto \langle \bot, \{\langle \mathbf{cl}^{\ominus}\ d_0, 0\rangle, \langle \mathbf{cl}^{\ominus}\ d_1, 1\rangle\}\rangle \\ &\quad\ d_0 \mapsto \langle \bot, \emptyset \rangle, \\ &\quad\ d_1 \mapsto \langle \bot, \emptyset \rangle\} \end{aligned}$$

where we have not named the results of $(\mathsf{sub1}\ a_1)$ and $(\mathsf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6)$ to avoid clutter. The direct analysis of M begins by applying both closures bound to f to the abstract value of 3. The analysis then combines the results of these two applications associating $\langle 0, \emptyset \rangle \sqcup \langle 1, \emptyset \rangle = \langle \top, \emptyset \rangle$ with a_1. As a consequence, the analysis loses all information about the value of a_2. In contrast, the analysis of the CPS version:

$$(f\ 3\ (\lambda a_1.\mathcal{F}_k[\![(\mathbf{let}\ (a_2\ (\mathbf{if0}\ a_1\ 5\ (\mathbf{if0}\ (\mathsf{sub1}\ a_1)\ 5\ 6)))\ a_2)]\!]))$$

duplicates the continuation $(\lambda a_1. \ldots)$ when evaluating the application of each of the closures bound to f. The analysis determines that the value of a_2 is $\langle 5, \emptyset, \emptyset \rangle$ along each execution path and hence improves on the analysis of the source program. □

6.3.2 Direct versus Semantic-CPS

The character of the relationship between the direct and the semantic-CPS analysis depends on whether the following condition holds.

Definition 6.1 *(Distributivity)* We call an analysis distributive[20] if for all κ, A_i, and n:

$$\begin{array}{c}\langle \kappa, \bigsqcup_{i=1,n} A_i\rangle \; appr^{\ominus} \; A_f \\ \text{if and only if } \langle \kappa, A_1\rangle \; appr^{\ominus} \; B_1 \; \ldots \; \langle \kappa, A_n\rangle \; appr^{\ominus} \; B_n \\ \text{and } A_f = \bigsqcup_{i=1,n} B_n.\end{array}$$

When the *Distributivity* condition does not hold, *e.g.*, for constant propagation [54, 60], the semantic-CPS data flow analyzer may gain information by duplicating the continuation along every execution path as in the right hand side of the condition. Otherwise, the semantic-CPS analysis is identical to the direct analysis of the program.

Theorem 6.7 Let $M \in \Lambda$, then $\langle M, \kappa, \sigma\rangle \; \mathcal{C}^{\ominus} \; A_1$ if and only if:

- $\langle M, \sigma\rangle \; \mathcal{M}^{\ominus} \; A_2$, and $\langle \kappa, A_2\rangle \; appr^{\ominus} \; A_3$, and $A_1 \sqsubseteq A_3$, or
- if *Distributivity* holds, $\langle M, \sigma\rangle \; \mathcal{M}^{\ominus} \; A_2$, and $\langle \kappa, A_2\rangle \; appr^{\ominus} \; A_1$.

Proof Idea The proof is by induction on the height of the derivation of A_1. The *Distributivity* condition is used only at conditionals when both branches are evaluated, and at $appk^{\ominus}$ when more than one closure is applied. □

6.3.3 Syntactic-CPS versus Semantic-CPS

The semantic-CPS analyzer may yield more precise results than the syntactic-CPS analyzer since the latter may confuse the continuations collected at a given label (cf. Theorem 6.5).

Theorem 6.8 Let $M \in \Lambda$, then $\langle M, \mathbf{nil}, \sigma\rangle \; \mathcal{C}^{\ominus}$

$$A_1 \text{ if and only if } \langle \mathcal{F}_k[\![M]\!], \delta^{\ominus}(\sigma)[k := \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle]\rangle \; \mathcal{M}_c^{\ominus} \; A_2$$

where $\delta^{\ominus}(A_1) \sqsubseteq A_2$.

---

[20]In the traditional framework, the lattice is usually inverted and the *Distributivity* condition is stated using the greatest lower bound $\sqcap$. In our case, the condition should be called *Continuity* but, to avoid confusion, we use the standard terminology.

Proof Intuitively, the proof is the abstract version of the proof in the previous section. The crucial point is that no new location is created for each continuation. Thus, the locations $k_1, k_2, \ldots$ may refer to more than one continuation. We strengthen the statement as follows: Let $M \in \Lambda$, and let:

$$\begin{array}{rcl} E_i & = & (\mathbf{let}\ (u_i\ [\,])\ N_i) \\ \kappa(h, i, \ldots, j, \top) & = & E_h :: E_i :: \ldots :: E_j :: \mathbf{nil} \\ \mathbf{co}(i, j) & = & \langle \bot, \emptyset, \{\langle \mathbf{co}^{\ominus}\ u_i, \mathcal{F}_{k_j}[\![N_i]\!]\rangle\}\rangle \\ \sigma(h, i, \ldots, j, \top) & = & s[k_h := \mathbf{co}(h, i), \ldots, k_j := \mathbf{co}(j, \top), k_\top := \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle] \end{array}$$

then:

$$\langle M, \kappa(h, \ldots, \top), \sigma\rangle\ \mathcal{C}^{\ominus}\ \langle u_1, \sigma_1\rangle \quad \text{if and only if} \quad \langle \mathcal{F}_{k_h}[\![M]\!], \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ \langle u_2, \sigma_2\rangle$$

where $\delta^{\ominus}(\sigma)(h, \ldots, \top) \sqsubseteq \sigma'$, $\delta^{\ominus}(u_1) \sqsubseteq u_2$ and $\delta^{\ominus}(\sigma_1)(h, \ldots, \top) \sqsubseteq \sigma_2$.

The proof proceeds by induction on the height of the derivation of A_1. For the base case, M is a value V and the continuation is the initial one:

$$\frac{u = \phi^{\ominus}(V, \sigma) \quad \langle \mathbf{nil}, \langle u, \sigma\rangle\rangle\ \mathit{appr}^{\ominus}\ \langle u, \sigma\rangle}{\langle V, \kappa(\top), \sigma\rangle\ \mathcal{C}^{\ominus}\ \langle u, \sigma\rangle}$$

$$\frac{u' = \phi_c^{\ominus}(\mathcal{V}[\![V]\!], \sigma') \quad \dfrac{\cdots}{\langle (K_\top \sqcup \langle \bot, \emptyset, \{\mathbf{stop}\}\rangle), \langle u', \sigma'\rangle\rangle\ \mathit{appr}_c^{\ominus}\ \langle u' \sqcup u'', \sigma' \sqcup \sigma''\rangle}}{\langle (k_\top\ \mathcal{V}[\![V]\!]), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ \langle u' \sqcup u'', \sigma' \sqcup \sigma''\rangle}$$

The result follows since $\delta^{\ominus}(u) \sqsubseteq u'$.

For the inductive case, we proceed by cases of M:

- $M = V$:

$$\frac{u = \phi^{\ominus}(V, \sigma) \quad \dfrac{\langle N_h, \kappa(i, \ldots, \top), \sigma[u_h := \sigma(u_h) \sqcup u]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle \kappa(h, i, \ldots, \top), \langle u, \sigma\rangle\rangle\ \mathit{appr}^{\ominus}\ A}}{\langle V, \kappa(h, i, \ldots, \top), \sigma\rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u' \sqsupseteq \delta^{\ominus}(u) \quad \dfrac{\cdots \quad \langle \mathcal{F}_{k_i}[\![N_h]\!], \sigma'[u_h := \sigma'(u_h) \sqcup u']\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle K_h \sqcup \mathbf{co}(h, i), \langle u', \sigma'\rangle\rangle\ \mathit{appr}_c^{\ominus}\ A' \sqcup A''}}{\langle (k_h\ \mathcal{V}[\![V]\!]), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A' \sqcup A''}$$

  The result follows by induction.

- $M = (\textbf{let}\ (x\ V)\ N)$:

$$\frac{u = \phi^{\ominus}(V,\sigma) \quad \langle N, \kappa(h,\ldots,\top), \sigma[x := \sigma(x) \sqcup u]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle(\textbf{let}\ (x\ V)\ N), \kappa(h,\ldots,\top), \sigma\rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u' \sqsupseteq \delta^{\ominus}(u) \quad \langle \mathcal{F}_{k_h}[\![N]\!], \sigma'[x := \sigma'(x) \sqcup u']\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle(\textbf{let}\ (x\ \mathcal{V}[\![V]\!])\ \mathcal{F}_{k_h}[\![N]\!]), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

The result follows by induction.

- $M = (\textbf{let}\ (u_g\ (V_1\ V_2))\ N_g)$:

$$\frac{u_1 = \phi^{\ominus}(V_1,\sigma) \quad u_2 = \phi^{\ominus}(V_2,\sigma) \quad \langle u_1, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}{\langle(\textbf{let}\ (u_g\ (V_1\ V_2))\ N_g), \kappa(h,\ldots,\top), \sigma\rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_1' \sqsupseteq \delta^{\ominus}(u_1) \quad u_2' \sqsupseteq \delta^{\ominus}(u_2) \quad \langle u_1', u_2', \textbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}{\langle(\mathcal{V}[\![V_1]\!]\ \mathcal{V}[\![V_2]\!]\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])), \sigma'\rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

Let the closure component of the abstract value u_1 be CL. Since $u_1' \sqsupseteq \delta^{\ominus}(u_1)$, the closure component of u_1' must at least include CL. Since the results of all the calls are combined, it suffices to establish the result for each closure $cl \in CL$ separately:

  - $cl = \textbf{inc}$, $u_2 = \langle n, \ldots\rangle$:

$$\frac{\dfrac{\langle N_g, \kappa(h,\ldots,\top), \sigma[u_g := \sigma(u_g) \sqcup \langle \mathit{add1}^{\ominus}(n), \emptyset\rangle]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle \kappa(g,h,\ldots,\top), \langle\langle \mathit{add1}^{\ominus}(n), \emptyset\rangle, \sigma\rangle\rangle\ \mathit{appr}^{\ominus}\ A}}{\langle \textbf{inc}, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}$$

$$\frac{\dfrac{\langle \mathcal{F}_{k_h}[\![N_g]\!], \sigma'[u_g := \sigma'(u_g) \sqcup \langle \mathit{add1}^{\ominus}(n), \emptyset, \emptyset\rangle]\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle \textbf{co}(g,h), \langle\langle \mathit{add1}^{\ominus}(n), \emptyset, \emptyset\rangle, \sigma'\rangle\rangle\ \mathit{appr}_c^{\ominus}\ A'}}{\langle \textbf{inck}, u_2', \textbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}$$

  The result follows by induction.

  - $cl = \textbf{dec}$: similar to the previous case.
  - $cl = \langle \textbf{cl}^{\ominus}\ z, N_z\rangle$:

$$\frac{\langle N_z, \kappa(g,h,\ldots,\top), \sigma[z := \sigma(z) \sqcup u_2]\rangle\ \mathcal{C}^{\ominus}\ A}{\langle\langle \textbf{cl}^{\ominus}\ z, N_z\rangle, u_2, \kappa(g,h,\ldots,\top), \sigma\rangle\ \mathit{appk}^{\ominus}\ A}$$

$$\frac{\langle \mathcal{F}_{k_g}[\![N_z]\!], \sigma'[z := \sigma'(z) \sqcup u_2', k_g := \sigma'(k_g) \sqcup \textbf{co}(g,h)]\rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle\langle \textbf{cl}^{\ominus}\ z k_g, \mathcal{F}_{k_g}[\![N_z]\!]\rangle, u_2', \textbf{co}(g,h), \sigma'\rangle\ \mathit{app}_c^{\ominus}\ A'}$$

  The result follows by induction.

- $M = (\textbf{let}\ (u_g\ (\textbf{if0}\ V_0\ M_1\ M_2))\ N_g)$:

$$\frac{u_0 = \phi^{\ominus}(V_0, \sigma) = \langle 0, \emptyset \rangle \quad \langle M_1, \kappa(g, h, \ldots, \top), \sigma \rangle\ \mathcal{C}^{\ominus}\ A}{\langle (\textbf{let}\ (u_g\ (\textbf{if0}\ V_0\ M_1\ M_2))\ N_g), \kappa(h, \ldots, \top), \sigma \rangle\ \mathcal{C}^{\ominus}\ A}$$

$$\frac{u_0' = \langle 0, \emptyset, \emptyset \rangle \quad \langle \mathcal{F}_{k_g}[\![M_1]\!], \sigma'[k_g := \sigma'(k_g) \sqcup \textbf{co}(g, h)] \rangle\ \mathcal{M}_c^{\ominus}\ A'}{\langle (\textbf{let}\ k_g\ (\lambda u_g.\mathcal{F}_{k_h}[\![N_g]\!])\ (\textbf{if0}\ \mathcal{V}[\![V_0]\!]\ \mathcal{F}_{k_g}[\![M_1]\!]\ \mathcal{F}_{k_g}[\![M_2]\!])), \sigma' \rangle\ \mathcal{M}_c^{\ominus}\ A'}$$

Since in general, $u_0' \sqsupseteq \delta^{\ominus}(u_0)$, the syntactic-CPS interpreter may also evaluate both branches. Both cases follow by induction. The cases when the semantic-CPS interpreter evaluates M_2 or when it evaluates both branches are similar.

$\square$

6.4 Discussion of the Results

In summary, the theorems show that:

1. The syntactic-CPS analyzer:
   - may confuse some continuations, and hence may consider a path that corresponds to a "false return",
   - duplicates the analysis of continuations along every execution path, and hence may gather more information than the direct analyzer in non-distributive analyses. This result follows from the combination of Theorems 6.7 and 6.8.
2. The semantic-CPS analyzer does not suffer from the false return problem and increases the collected information in non-distributive analyses by the duplication of the analysis of continuations.

In the remainder of this section, we discuss each of the properties of the CPS analyzers in detail.

6.4.1 False Returns

In practice, many analyses do indeed confuse continuations when applied to CPS programs. For example, Shivers's 0CFA analysis of CPS programs [92] merges distinct control paths unnecessarily. Shivers did not relate the problem to CPS but his example [92, p 33] is essentially the example for Theorem 6.5.

Given our result, we can explain how the CPS transformation confuses some data flow analyzers that associate (approximate) information with program points. Because the CPS transformation reifies the continuation to a value that the program manipulates explicitly, the analysis of a CPS program is obligated to collect, at each variable k, the set of continuations that k may refer to during the execution of the program. Thus, when considering a return, *i.e.*, a call $(k\ W)$, the analysis applies each of the continuations bound to k and merges the results. In contrast, the analysis of the source program and the semantic-CPS analysis do not collect continuations, but only consider the *current* continuation at any program point.

6.4.2 Duplication

The gain of information in semantic-CPS analyzers is folklore knowledge. Nielson [72] proved that, for a small imperative language, the semantic-CPS analysis computes the MOP (meet over all paths) solution and the direct analysis computes the less precise MFP (maximum fixed point) solution; Filho and Burn [34] improved the abstract interpretations of typed call-by-name languages using the CPS transformation. Our result suggests that the gain in all cases is entirely due to the duplication of the analysis of the continuation along different execution paths. In the remainder of this section, we consider the impact of this duplication on the computability and cost of the analysis.

Intuitively, the difference between the direct semantics and the CPS analyses is that the former merges all the values of an expression before analyzing the continuation and the latter apply the continuation to each of the values of an expression and merge the results. Therefore, the duplication of the analysis of the continuation depends on the number of values an expression may have.

Thus far, every expression in our language had only a finite number of values: the analysis of a conditional expression may proceed along two paths, and the analysis of a procedure call may proceed along some finite number of paths, one for each abstract closure that the term in function position evaluates to. Consequently, at each conditional and at each call site, the continuation may be duplicated along each of the possible paths, at an overall *exponential* cost in the analysis.

In a realistic language, the duplication of the continuation causes the computation of the result of the CPS analysis to become undecidable. To illustrate this point, we assume an extension of the language with an explicit looping construct and a sufficiently rich set of primitives.

Let the construct loop be an infinite loop whose exact collecting semantics returns the infinite set of values $\{0, 1, 2, \ldots\}$.[21] The extensions of the direct and semantic-CPS analyzers are:

$$\frac{u_i = \langle i, \emptyset\rangle \quad A_i = \langle u_i, \sigma\rangle}{\langle \mathsf{loop}, \sigma\rangle \ \mathcal{M}^{\ominus}\ (\bigsqcup_{i=0,\infty} A_i)} \qquad \frac{u_i = \langle i, \emptyset\rangle \quad A_i = \langle u_i, \sigma\rangle \quad \langle \kappa, A_i\rangle \ \mathit{appr}^{\ominus}\ B_i}{\langle \mathsf{loop}, \kappa, \sigma\rangle \ \mathcal{C}^{\ominus}\ (\bigsqcup_{i=0,\infty} B_i)}$$

In the direct interpreter, each u_i is an abstract number $\langle i, \emptyset\rangle$ and the least upper bound of the set $\{u_i \mid i \geq 0\}$ is $\langle \top, \emptyset\rangle$. In the semantic-CPS case, the computation of $\bigsqcup_{i=0,\infty} B_i$ is undecidable. The proof is an adaptation of Kam and Ullman's [54] that, given an arbitrary monotone framework, it is undecidable to compute the MOP solution for each program:

> Given a Turing machine H, we construct the continuation κ such that:
>
> $$\langle \kappa, \langle\langle n, \emptyset\rangle, \sigma\rangle\rangle \ \mathit{appr}^{\ominus}\ \langle\langle n', \emptyset\rangle, \sigma\rangle$$
>
> where:
>
> - $n' = n$ if n is $\perp$ or $\top$, and
> - if $n = i$ for some natural number i, then the continuation simulates the Turing machine H for i steps and returns $n' = 0$ if H halts in less than i steps and $n' = 1$ if H fails to terminate after i steps.[22]
>
> Given the above construction, each B_i in the set whose least upper bound we want to compute is either $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ (indicating H halts in less than i steps), or $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ (indicating H does not halt in less than i steps). Therefore, the least upper of the set is either $\langle\langle 0, \emptyset\rangle, \sigma\rangle$, $\langle\langle 1, \emptyset\rangle, \sigma\rangle$, or $\langle\langle \top, \emptyset\rangle, \sigma\rangle$. In the first case, all entries are $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ indicating that H terminates in 0 steps. In the second case, all entries are $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ indicating that H does not halt in any number of steps. In the last case, some of the entries (the first j entries) are $\langle\langle 1, \emptyset\rangle, \sigma\rangle$ and some (all remaining entries) are $\langle\langle 0, \emptyset\rangle, \sigma\rangle$ indicating that H halts after j steps. Thus, we can determine whether the Turing machine H halts or not by computing the least upper bound of the set $\{B_i \mid i \geq 0\}$, which implies that the latter operation must be undecidable.

---

[21]The construct loop corresponds to the following program fragment '$x := 0;$ **while** true $x := x + 1$'.

[22]This step assumes that the abstract operations are rich enough to express any finite sequence of steps of a given Turing machine. We assume that the language is extended with powerful enough primitives such as string concatenation, division, etc.

It is easy to modify the argument to prove the un-decidability of the semantic-CPS analysis for other languages assuming they have a sufficiently rich set of primitive operations.

6.5 Summary

In conclusion, a *practical* analysis based on the CPS transformation should not perform any duplication when the analysis is distributive since the duplication would not yield more precise answers. In non-distributive cases, a CPS analysis should limit the amount of duplication for both computability and efficiency reasons. In the case when the analysis of a CPS program does not perform any duplication, the net effect of transforming the program to CPS is to obscure the fact that there is only *one* control stack at any point during a computation. Alternatively, a direct analysis that allows some duplication, *e.g.*, in-lining some procedures in flow-sensitive analyses, would be as satisfactory as a practical CPS-based analysis.

1. The results of a direct analysis of a source program are *incomparable* to the results of an analysis of the equivalent CPS program. In other words, the translation of the source program to a CPS version may increase or decrease static information. The gain of information occurs in non-distributive analyses and is solely due to the *duplication* of the analysis of the continuation. The loss of information is due to the confusion of distinct procedure returns.

2. The analyzer based on the continuation semantics produces more accurate results than both direct analyzers, but again only in non-distributive analyses due to the *duplication* of continuations along every execution path. However, when the analyzer explicitly accounts for looping constructs, the results of the semantic-CPS analysis are no longer computable.

In view of these results, we argue that, in practice, a direct data flow analysis that relies on some amount of duplication would be as satisfactory as a CPS analysis.

Chapter 7

Towards A Synthesis of Direct and CPS Compilation

The main contribution of this dissertation is a clear and formal connection between the two major strategies for compiling higher-order programming languages. The first strategy uses the CPS transformation to generate the main intermediate representation whereas the second relies on other *ad hoc* techniques. Our research suggests a compilation strategy based on normalizing source programs using the administrative call-by-value reductions. This strategy combines the best aspects of both the direct and CPS paradigms:

- Like CPS compilers, the generation of the intermediate representation is systematic. This phase can benefit from the extensive studies of algorithms for CPS conversion by adapting the most efficient CPS algorithms to the new transformation.

- The intermediate representation can be simplified and optimized with a theory that is as powerful as the canonical calculus used by CPS compilers, thus improving on traditional direct compilers.

- Like direct compilers, we can generate code in a straightforward manner without having to go through the contortions of "inverting" the CPS intermediate representation on the fly by tagging continuations and optimizing their manipulation by keeping them in global registers.

- Finally, unlike CPS code, the intermediate representation does not confuse certain classes of data flow analyses, and the analyses can be tuned to control the tradeoff between precision and cost.

In summary, we argue that the most useful aspect of the CPS transformation in compilers is that it converts the high-level functional source program to an explicit sequence of elementary computation steps, whose semantics is simple and well-understood. This same goal can be achieved more clearly by eliminating the administrative redexes from the original source program. This normalization can be done as systematically and as efficiently as conversion to CPS, and is convenient for code generation and data-flow analysis.

The most natural continuation of this work is to produce a complete Scheme compiler. It is quite simple to write a naïve compiler by using a simplistic back end that generates un-optimized code. However, given the number of respectable Scheme compilers that are in existence, writing yet-another-compiler is not an end in itself. The more important (and more challenging) goal is to extend the theory developed in this dissertation to encompass all of the phases of the compiler such as data representations, tail-call optimizations, instruction scheduling, etc. This comprehensive theory of compilation would give more insights in the process of compilation by formalizing the practices of compiler writers.

Bibliography

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass. (1985).

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

[3] Appel, A. *Compiling with Continuations.* Cambridge University Press (1992).

[4] 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.

[5] Baker, H. CONS should not CONS its arguments, part II: Cheney on the M.T.A. (January 1994). Unpublished manuscript.

[6] 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).

[7] 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.

[8] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985) 637–655.

[9] 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.

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

[11] Clinger, W. The Scheme 311 compiler: An exercise in denotational semantics. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1984) 356–364.

[12] Clinger, W. and Hansen, L. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994). To Appear.

[13] Clinger, W. and Rees, J. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4, 3 (1991) 1–55.

[14] 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.

[15] Consel, C. and Danvy, O. For a better support of static data flow. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1991) 496–519.

[16] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (1977) 238–252.

[17] 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.

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

[19] Danvy, O. *Three Steps for the CPS Transformation.* Technical Report CIS-92-2, Kansas State University (1992).

[20] 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.

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

[22] 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.

[23] Demers, A. and Donahue, J. Making variables abstract: An equational theory for Russell. In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages* (1983) 59–72.

[24] 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.

[25] Felleisen, M. λ-v-CS: An extended λ-calculus for Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1988) 72–84.

[26] Felleisen, M. and Friedman, D.P. A calculus for assignments in higher-order languages. In *Conference Record of the 14th ACM Symposium on Principles of Programming Languages* (1987) 314–325.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Fessenden, C., Clinger, W., Friedman, D. P., and Haynes, C. T. *Scheme 311 Version 4 Reference Manual.* Computer Science Technical Report 137, Indiana University, Bloomington, Indiana (February 1983).

[32] Field, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis (preliminary report). In *ACM Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report RR-909, Yale University (1992) 98–107.

[33] Filho, J. Muylaert. Improving abstract interpretations with CPS-translation. (1993). Unpublished Manuscript.

[34] Filho, J. Muylaert and Burn, G. Continuation passing transformation and abstract interpretation. In Burn, G., Gay, S., and Ryan, M., editors, *Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods* (1993).

[35] Filinski, A. Representing monads. In *Conference Record of the 21th ACM Symposium on Principles of Programming Languages* (1994) 446–457.

[36] Filinski, A. Representing monads (revised version). (1994). In preparation.

[37] Fischer, M. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions About Programs*, Sigplan Notices, 7, 1 (1972) 104–109. Revised version in *Lisp and Symbolic Computation*, **6**, 3/4, (1993) 259–287.

[38] 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) 237–247.

[39] Fradet, P. and Métayer, D. Le. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13, 1 (January 1991) 21–51.

[40] 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.

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

[42] Friedman, H. Equality between functionals. In Parikh, R., editor, *Logic Colloquium*, Springer Verlag, Berlin (1973) 22–37.

[43] G. D. Plotkin, G.D. LCF considered as a programming language. *Theoretical Computer Science*, 5 (1977) 223–255.

[44] 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.

[45] Gateley, J. and Duba, B.F. Call-by-value combinatory logic and the lambda-value calculus. In Brookes, S., editor, *Proceedings of the Conference on the Mathematical Foundations of Programing Semantics*, Lecture Notes in Computer Science, 517, Springer Verlag (1991).

[46] 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.

[47] Gunter, Carl A. *Semantics of Programming Languages. Foundations of Computing*, MIT Press, Cambridge, MA (1992).

[48] Hatcliff, J. and Danvy, O. A generic account of continuation-passing styles. In *Conference Record of the 21th ACM Symposium on Principles of Programming Languages* (1994).

[49] 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.

[50] 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.

[51] Hoare, C., Hayes, I., Jifeng, H., Morgan, C., Roscoe, A., Sanders, J., Sorensen, I., Spivey, J., and Sufrin, B. Laws of programming. *Communications of the ACM*, 30, 8 (1987) 672–686.

[52] Hofmann, M. Sound and complete axiomatisations of call-by-value control operators. In *Proceedings of the 5th Conference on Category Theory and Computer Science* (1993). To Appear.

[53] Hudak, P. and Young, J. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13, 2 (April 1991) 269–290.

[54] Kam, J.B. and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica*, 7 (1977) 305–317.

[55] 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.

[56] 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.

[57] Lawall, J. and Danvy, O. Separating stages in the continuation-passing transform. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (1993) 124–136.

[58] Lawall, J. and Danvy, O. Continuation-based partial evaluation. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994). To Appear.

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

[60] Marlowe, T.J. and Ryder, B.G. Properties of data flow frameworks: A unified model. *Acta Informatica* (1990).

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

[62] Mason, I.A. and Talcott, C. L. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105, 2 (1992) 167–215. Preliminary version in Proceedings of the 4th IEEE Symposium on Logic in Computer Science 1989.

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

[64] Meyer, A. and Riecke, J. Continuations may be unreasonable. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1988) 63–71.

[65] 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.

[66] Milner, R. Functions as processes. *Mathematical Structures in Computer Science*, 2 (1992) 119–141.

[67] 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.

[68] Moggi, E. Notions of computation and monads. *Information and Computation*, 93 (1991) 55–92.

[69] Morris, L. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6, 3/4 (1993) 249–256. Original manuscript dated 1970.

[70] Murthy, C. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell (1990).

[71] Murthy, C. An evaluation semantics for classical proofs. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science* (1991).

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

[73] Odersky, M., Rabin, D., and Hudak, P. Call by name, assignment, and the lambda calculus. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (January 1993) 43–56.

[74] Okasaki, C., Lee, P., and Tarditi, D. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7, 1 (1994) 57–81.

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

[76] Plotkin, G. D. Completeness for the continuous type frame. Unpublished Manuscript.

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

[78] Reynolds, J. C. On the relation between direct and continuation semantics. In *Proceedings of the International Colloquium on Automata, Languages and Programming* (1974) 141–156.

[79] Reynolds, J. C. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 3/4 (1993) 233–247.

[80] Riecke, J. Delimiting the scope of effects. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1993) 146–155.

[81] Riecke, J. Statman's 1-section theorem. *Information and Computation* (1994). To appear. Also Technical Report MS-CIS-92-03, University of Pennsylvania, 1992.

[82] Rozas, G. *Liar, An Algol-like Compiler for Scheme*. Master's thesis, MIT Department of Electrical Engineering and Computer Science (1984).

[83] 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.

[84] Sabry, A. and Felleisen, M. *Reasoning about Programs in Continuation-Passing Style*. Technical Report 92-180, Rice University (1992).

[85] Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6, 3/4 (1993) 289–360. Also in Proceedings of the ACM Conference on Lisp and Functional Programming, 1992, and Technical Report 92-180, Rice University.

[86] Sabry, A. and Felleisen, M. Is continuation-passing useful for data flow analysis? In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1994). To appear.

[87] Sabry, A. and Felleisen, M. *Is Continuation-Passing Useful for Data Flow Analysis?* Technical Report TR94-223, Rice University (1994).

[88] Sabry, A. and Field, J. Reasoning about explicit and implicit representations of state. In *ACM Sigplan Workshop on State in Programming Languages*, Technical Report RR-968, Yale University (1993) 17–30.

[89] Scott, D. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems* ( ) 169–176. Also Technical Monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group, 1970.

[90] Scott, D. and Strachey, C. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata* (1971). Also Technical Monograph PRG-6, Oxford University, Computing Laboratory, Programming Research Group.

[91] Sethi, R. and Tang, A. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27, 3 (1980) 580–597.

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

[93] Sitaram, D. Handling control. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1993) 147–155.

[94] Sitaram, D. and Felleisen, M. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3, 1 (1990) 67–100.

[95] 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.

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

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

[98] Stoy, J. The congruence of two programming language definitions. *Theoretical Computer Science*, 13 (1981) 151–174.

[99] Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Languages.* The MIT Press, Cambridge, Mass. (1981).

[100] Strachey, C. Towards a formal semantics. In Steel, T.B., editor, *Formal Language Description Languages for Computer Programming*, North-Holland (1966) 197–220.

[101] 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).

[102] Swarup, V., Reddy, U., and Ireland, E. Assignments for applicative languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1991) 192–214.

[103] 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).

[104] 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.

[105] Wright, A. and Duba, B. F. Pattern matching for Scheme. Unpublished manuscript, 1993. Available as World Wide Web URL "ftp://cs.rice.edu/public/wright/match.ps.Z".